

CS241 System Programming Wikibook

Contents

1	Introduction	5
2	Deadlock	7
2.1	Resource Allocation Graphs	7
2.2	Coffman conditions	8
2.3	Approaches to solving deadlock	9
2.4	Dining Philosophers	10
2.5	Viable Solutions	12
2.6	Topics	13
2.7	Questions	13
3	Appendix	15
3.1	The Hitchhiker's Guide to Debugging C Programs	15
3.2	Valgrind	16
3.3	GDB	19
3.4	Life in the terminal	20
3.5	System Programming Jokes	20

Chapter 1

Introduction

`./introduction`

Chapter 2

Deadlock

No, you can't always get what you want
You can't always get what you want
You can't always get what you want
But if you try sometime you find
You get what you need

The philosophers Jagger & Richards

Deadlock is defined as when the system cannot make and forward progress. In a lot of systems, Deadlock is just avoided by ignore the entire concept (Silberschatz et al., 2006, P237). Have you heard about turn it on and off again? That is partly because of this. For products where the stakes are low when you deadlock (User Operating Systems, Phones), it may be more efficient not not keep track of all of the allocations in order to keep deadlock from happening. But in the cases where "failure is not an option" - Apollo 13, you need a system that tracks deadlock. Take the Apollo 13 module. It may have not failed because of deadlock, but probably wouldn't be good to restart the system on liftoff.

Mission critical operating systems need this guarentee formally because playing the odds with people's lives isn't a good idea. Okay so how do we do this? We model the problem. Even though it is a common statistical phrase that all models are wrong, the more accurate the model is to the system that we are working with the better chance that it'll work better.

Resource Allocation Graphs

A resource allocation graph tracks which resource is held by which process and which process is waiting for a resource of a particular type. It is very powerful and simple tool to illustrate how interacting processes can deadlock. If a process is *using* a resource, an arrow is drawn from the resource node to the process node. If a process is *requesting* a resource, an arrow is drawn from the process node to the resource node.

If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will deadlock. For example, if process 1 holds resource A, process 2 holds resource B and process 1 is waiting for B and process 2 is waiting for A, then process 1 and 2 process will be deadlocked.

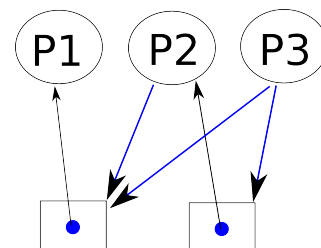


Figure 2.1: Resource allocation graph

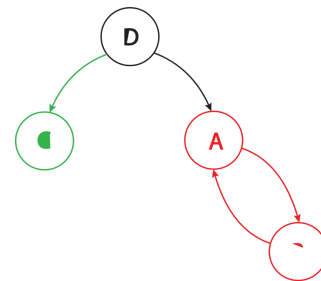
Here's another example, that shows Processes 1 and 2 acquiring resources 1 and 2 while process 3 is waiting to acquire both resources. In this example there is no deadlock because there is no circular dependency.

Deadlock!

A lot of times, we don't know the specific order that a resource may be acquired so we can draw the graph directed.

As a possibility matrix. Then we can draw arrows and see if there is a directed version that would lead us to a deadlock.

Consider the following resource allocation graph. Assume that the processes ask for exclusive access to the file. If you have a bunch of processes running and they request resources and the operating system ends up in this state, you deadlock! You may not see this because the operating system may **preempt** some processes breaking the cycle but there is still a change that your three lonely processes could deadlock. You can also make these kind of graphs with make and rule dependencies (with our parmake MP for example).



Coffman conditions

There are four *necessary* and *sufficient* conditions for deadlock. These are known as the Coffman conditions ?

Figure 2.2: Colorful Deadlock

- Mutual Exclusion: no two processes can obtain a resource at the same time.
- Circular Wait: there exists a cycle in the Resource Allocation Graph, or there exists a set of processes $\{P_1, P_2, \dots\}$ such that P_1 is waiting for resources held by P_2 , which is waiting for P_3, \dots , which is waiting for P_1 .
- Hold and Wait: a process once obtaining a resource does not let go.
- No pre-emption: nothing can force the process to give up a resource.

If you break any of them, you cannot have deadlock! Consider the scenario where two students need to write both pen and paper and there is only one of each. Breaking mutual exclusion means that the students share the pen and paper. Breaking circular wait could be that the students agree to grab the pen then the paper. As a proof by contradiction, say that deadlock occurs under the rule and the conditions. Without loss of generality, that means a student would have to be waiting on a pen while holding the paper and the other waiting on a pen and holding the paper. We have contradicted ourselves because one student grabbed the paper without grabbing the pen, so deadlock must not be able to occur. Breaking hold and wait could be that the students try to get the pen and then the paper and if a student fails to grab the paper then they release the pen. This introduces a new problem called *livelock* which will be discussed latter. Breaking preemption means that if the two students are in deadlock the teacher can come in and break up the deadlock by giving one of the students one the held on items or tell both students to put the items down.

Livelock relates to deadlock but it is not exactly deadlock. Consider the breaking hold and wait solution as above. Though deadlock is avoided, if we pick up the same device (a pen or the paper) again and again in the exact same pattern, neither of us will get any writing done. More generally, livelock happens when the process looks like it is executing but no meaningful work is done. Livelock is generally

harder to detect because the processes generally look like they are working to the outside operating system whereas in deadlock the operating system generally knows when two processes are waiting on a system wide resource. Another problem is that there are necessary conditions for livelock (i.e. deadlock does not occur) but not sufficient conditions – meaning there is no set of rules where livelock has to occur. You must formally prove in a system by what is known as an invariant. One has to enumerate

Approaches to solving deadlock

Ignoring deadlock is the most obvious approach that started the chapter out detailing. Quite humorously, the name for this approach is called the ostrich algorithm. Though there is no apparent source, the idea for the algorithm comes from the concept of an ostrich sticking its head in the sand. When the operating system detects deadlock, it does nothing out of the ordinary and hopes that the deadlock goes away. Now this is a slight misnomer because the operating system doesn't do anything *abnormal* – it is not like an operating system deadlocks every few minutes because it runs 100 processes all requesting shared libraries. An operating system still preempts processes when stopping them for context switches. The operating system has the ability to interrupt any system call, potentially breaking a deadlock scenario. The OS also makes some files read-only thus making the resource shareable. What the algorithm refers to is that if there is an adversary that specifically crafts a program – or equivalently a user who poorly writes a program – that deadlock could not be caught by the operating system. For everyday life, this tends to be fine. When it is not we can turn to the following method.

Deadlock detection allows the system to enter a deadlocked state. After entering, the system uses the information that it has to break deadlock. As an example, consider multiple processes accessing files. The operating system is able to keep track of all of the files/resources through file descriptors at some level either abstracted through an API or directly. If the operating system detects a directed cycle in the operating system file descriptor table it may break one process' hold through scheduling for example and let the system proceed. Why this is a popular choice in this realm is that there is no way of knowing which resources a program will select without running the program. This is an extension of Rice's Theorem ? that says that we cannot know any semantic feature without running the program (semantic meaning like what files it tries to open). So theoretically, it is sound. The problem then gets introduced that we could reach a livelock scenario if we preempt a set of resources again and again. The way around this is mostly probabilistic. The operating system chooses a random resource to break hold and wait. Now even though a user can craft a program where breaking hold and wait on each resource will result in a livelock, this doesn't happen as often on machines that run programs in practice or the livelock that does happen happens for a couple of cycles. These kind of systems are good for products that need to maintain a non-deadlocked state but can tolerate a small chance of livelock for a short period of time. The following proof is **not required for our 241 related purposes but is included for concreteness**.

Proof: That livelock terminates with high probability. This meaning that for any probability level l we can produce a number of iterations n that the probability that the system is not livelocked after that state is at least l .

Let $L = \{p_1, p_2, p_3, \dots\}$ be an infinite set that has probabilities if we choose a resource that causes livelock in the i th iteration of the livelocked by breaking a random resource. Also let the set have the property that $\forall i > 0, \exists j > i, p_j < 1$ Meaning that for all elements after any given element that there is atleast one element in the future that has a probability of breaking deadlock. The probability that the system is not livelocked after n iterations is

$$P(n) = 1 - \prod_{i=1}^n p_i$$

Consider the new set $E = \{s_1, s_2, s_3, \dots\}$ obtained by selecting all non-zero elements. The set must be infinite by our sets property. It is easy to show that

$$\prod_{i=1}^n p_i = \prod_{i=1}^n s_i$$

And since

$$P(n) = 1 - \prod_{i=1}^n s_i$$

is a strictly monotonically decreasing function, it must pass the threshold for the probability level at some point. More rigorously

$$\begin{aligned} Y &= \sup E \\ \prod_{i=1}^n s_i &< Y^n \\ 1 - \prod_{i=1}^n s_i &> 1 - Y^n \\ P(n) &> 1 - Y^n \\ P(n) &> l \\ 1 - Y^n &> l \\ \log_Y(1 - l) &< n \end{aligned}$$

We have found the number of steps in the set E and since our set has the property that gaps between non-one elements is finite, we can reconstruct the number of iterations by picking an n that satisfies the E criteria and just adding the finite gaps together, which is what we needed to show. \square

Deadlock prevention is making sure that deadlock cannot happen, meaning that you break a Coffman condition. This works the best inside a single program and the software engineer making the choice to break a certain Coffman condition. Consider the Banker's Algorithm. It is another algorithm for deadlock avoidance. The whole implementation is outside the scope of this class, just know that there are more generalized algorithms for operating systems.

Dining Philosophers

The Dining Philosophers problem is a classic synchronization problem. Imagine I invite n (let's say 5) philosophers to a meal. We will sit them at a table with 5 chopsticks, one between each philosopher. A philosopher alternates between wanting to eat or think. To eat the philosopher must pick up the two chopsticks either side of their position. The original problem required each philosopher to have two forks, but one can eat with a single fork so we rule this out. However these chopsticks are shared with his neighbor.

Is it possible to design an efficient solution such that all philosophers get to eat? Or, will some philosophers starve, never obtaining a second chopstick? Or will all of them deadlock? For example, imagine each guest picks up the chopstick on their left and then waits for the chopstick on their right to be free. Oops - our philosophers have deadlocked! Each of the philosophers are essentially the same, meaning that each philosopher has the same

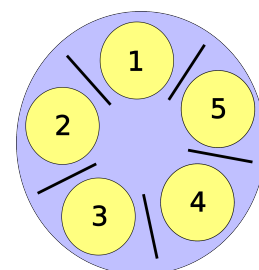


Figure 2.3: Dining Philosophers

instruction set based on the other philosopher ie you can't tell every even philosopher to do one thing and every odd philosopher to do another thing.

Failed Solutions

```
void* philosopher(void* forks){
    info phil_info = forks;
    pthread_mutex_t* left_fork = phil_info->left_fork;
    pthread_mutex_t* right_fork = phil_info->right_fork;
    while(phil_info->simulation){
        pthread_mutex_lock(left_fork);
        pthread_mutex_lock(right_fork);
        eat(left_fork, right_fork);
        pthread_mutex_unlock(left_fork);
        pthread_mutex_unlock(right_fork);
    }
}
```

This looks good but. What if everyone picks up their left fork and is waiting on their right fork? We have deadlocked the program. It is important to note that deadlock doesn't happen all the time and the probability that this solution deadlocks goes down as the number of philosophers goes up. What is really important to note is that eventually that this solution will deadlock, letting threads starve which is bad.

So now you are thinking about breaking one of the Coffman conditions. We have - Mutual Exclusion - No Preemption - Hold and wait - Circular Wait Well we can't have two philosophers use a fork at the same time, mutual exclusion is out of the picture. Let's break Hold and Wait!

```
void* philosopher(void* forks){
    info phil_info = forks;
    pthread_mutex_t* left_fork = phil_info->left_fork;
    pthread_mutex_t* right_fork = phil_info->right_fork;
    while(phil_info->simulation){
        pthread_mutex_lock(left_fork);
        pthread_mutex_lock(right_fork);
        eat(left_fork, right_fork);
        pthread_mutex_unlock(left_fork);
        pthread_mutex_unlock(right_fork);
    }
}
```

Now our philosopher picks up the left fork and tries to grab the right. If it's available, they eat. If it's not available, they put the left fork down and try again. No deadlock! But, there is a problem. What if all the philosophers pick up their left at the same time, try to grab their right, put their left down, pick up their left, try to grab their right. ... We have now livelocked our solution! Our poor philosophers are still starving, so let's give them some proper solutions.

Viable Solutions

The naive arbitrator solution is have one arbitrator (a mutex for example). Have each of the philosopher ask the arbitrator for permission to eat (i.e. trylock the mutex). This solution allows one philosopher to eat at a time. When they are done, another philosopher can ask for permission to eat. This prevents deadlock because there is no circular wait! No philosopher has to wait on any other philosopher. The advanced arbitrator solution is to implement a class that determines if the philosopher's forks are in the arbitrator's possession. If they are, they give them to the philosopher, let him eat, and take the forks back. This has the added bonus of being able to have multiple philosopher eat at the same time.

There are a lot of problems with these solutions. One is that they are slow and have a single point of failure or the arbitrator. Assuming that all the philosophers are good-willed, the arbitrator needs to be fair and be able to determine if a transaction would cause deadlock in the multi-arbitrator case. Further more in practical systems, the arbitrator tends to give forks to the same processes because of scheduling or pseudorandomness. Another important thing to note is that this prevents deadlock for the entire system. But in our model of dining philosophers, the philosopher has to release the lock themselves. Then you can consider the case of the malicious philosopher (let's say Decartes because of his Evil Demons) could hold on to the arbitrator forever. He would make forward progress and the system would make forward progress but there is no way of ensuring that each process makes forward progress without assuming something about the processes or having true preemption – meaning that a higher authority (let's say Steve Jobs) tells them to stop eating forcibly.

Leaving the Table (Stallings' Solution)

Why does the first solution deadlock? Well there are n philosophers and n chopsticks. What if there is only 1 philosopher at the table? Can we deadlock? No. How about 2 philosophers? 3? ... You can see where this is going. Stallings' ? solutions says to remove philosophers from the table until deadlock is not possible – think about what the magic number of philosophers at the table. The way to do this in actual system is through semaphores and letting a certain number of philosopher through. This has the benefit that multiple philosophers can be eating.

In the case that the philosophers aren't evil, this solution requires a lot of time-consuming context switching. There is also no reliable way to know the number of resources before hand. In the dining philosophers case, this is solved because everything is known but trying to specify and operating system where you don't know which file is going to get opened by what process leads you with a faulty solution. And again since semaphores are system constructs, they obey system timing clocks which means that the same processes tend to get added back into the queue again. Now if a philosopher becomes evil, then the problem becomes that there is no preemption. A philosopher can eat for as long as they want and the system will continue to function but that means the fairness of this solution can be low in the worst case. This works best with timeouts (or forced context switches) in order to ensure bounded wait times.

Partial Ordering (Dijkstra's Solution)

This is Dijkstra's solution (he was the one to propose this problem on an exam). Why does the first solution deadlock? Dijkstra thought that the last philosopher who picks up his left fork (causing the solution to deadlock) should pick up his right. He accomplishes it by number the forks $1..n$, and tells each of the philosopher to pick up his lower number fork.

Let's run through the deadlock condition again. Everyone tries to pick up their lower number fork first. Philosopher 1 gets fork 1, Philosopher 2 gets fork 2, and so on until we get to Philosopher n . They have to choose between fork 1 and n . fork 1 is already held up by philosopher 1, so they can't pick up that fork, meaning he won't pick up fork n . We have broken circular wait! Meaning deadlock isn't possible.

Problems:

- The philosopher needs to know the set of resources in order before grabbing any resources.
- You need to define a partial order to all of the resources.
- Prioritizes philosopher who have already eaten.

Advanced Solutions

There are many more advanced solutions a non-exhaustive list includes

- Clean/Dirty Forks (Chandra/Misra Solution)
- Actor Model (other Message passing models)

Topics

- Coffman Conditions
- Resource Allocation Graphs
- Dining Philosophers
- Failed DP Solutions
- Livelocking DP Solutions
- Working DP Solutions: Benefits/Drawbacks

Questions

- What are the Coffman Conditions?
- What do each of the Coffman conditions mean? (e.g. can you provide a definition of each one)
- Give a real life example of breaking each Coffman condition in turn. A situation to consider: Painters, Paint, Paintbrushes etc. How would you assure that work would get done?
- Be able to identify when Dining Philosophers code causes a deadlock (or not). For example, if you saw the following code snippet which Coffman condition is not satisfied?

```
// Get both locks or none
pthread_mutex_lock(a);
if(pthread_mutex_trylock( b )) { /* failure */
    pthread_mutex_unlock( a );
}
```

-
- The following calls are made

```
// Thread 1
pthread_mutex_lock(m1) // success
pthread_mutex_lock(m2) // blocks

// Thread 2
pthread_mutex_lock(m2) // success
pthread_mutex_lock(m1) // blocks
```

What happens and why? What happens if a third thread calls `pthread_mutex_lock(m1)` ?

- How many processes are blocked? As usual assume that a process is able to complete if it is able to acquire all of the resources listed below.
 - P1 acquires R1
 - P2 acquires R2
 - P1 acquires R3
 - P2 waits for R3
 - P3 acquires R5
 - P1 waits for R4
 - P3 waits for R1
 - P4 waits for R5
 - P5 waits for R1

(Draw out the resource graph!)

Bibliography

A. Silberschatz, P.B. Galvin, and G. Gagne. *OPERATING SYSTEM PRINCIPLES, 7TH ED.* Wiley student edition. Wiley India Pvt. Limited, 2006. ISBN 9788126509621. URL <https://books.google.com/books?id=WjvX0HmVT1MC>.

Chapter 3

Appendix

The Hitchhiker's Guide to Debugging C Programs

This is going to be a massive guide to helping you debug your C programs. There are different levels that you can check errors and we will be going through most of them. Feel free to add anything that you found helpful in debugging C programs including but not limited to, debugger usage, recognizing common error types, gotchas, and effective googling tips.

In-Code Debugging

Clean code

Make your code modular using helper functions. If there is a repeated task (getting the pointers to contiguous blocks in the malloc MP, for example), make them helper functions. And make sure each function does one thing very well, so that you don't have to debug twice.

Let's say that we are doing selection sort by finding the minimum element each iteration like so,

```
void selection_sort(int *a, long len){
    for(long i = len-1; i > 0; --i){
        long max_index = i;
        for(long j = len-1; j >= 0; --j){
            if(a[max_index] < a[j]){
                max_index = j;
            }
        }
        int temp = a[i];
        a[i] = a[max_index];
        a[max_index] = temp;
    }
}
```

Many can see the bug in the code, but it can help to refactor the above method into

```
long max_index(int *a, long start, long end);
void swap(int *a, long idx1, long idx2);
void selection_sort(int *a, long len);
```

And the error is specifically in one function. In the end, we are not a class about refactoring/debugging your code. In fact, most kernel code is so atrocious that you don't want to read it. But for the sake of debugging, it may benefit you in the long run to adopt some of these practices.

Asserts!

Use assertions to make sure your code works up to a certain point – and importantly, to make sure you don't break it later. For example, if your data structure is a doubly linked list, you can do something like `assert(node->size == node->next->prev->size)` to assert that the next node has a pointer to the current node. You can also check the pointer is pointing to an expected range of memory address, not null, `->size` is reasonable etc. The `NDEBUG` macro will disable all assertions, so don't forget to set that once you finish debugging. [assert link](#)

Here's a quick example with assert. Let's say that I'm writing code using `memcpy`

```
assert(!(src < dest+n && dest < src+n)); //Checks overlap
memcpy(dest, src, n);
```

This check can be turned off at compile time, but will save you **tons** of trouble debugging!

printfs

When all else fails, print like crazy! Each of your functions should have an idea of what it is going to do (ie `find_min` better find the minimum element). You want to test that each of your functions is doing what it set out to do and see exactly where your code breaks. In the case with race conditions, `tsan` may be able to help, but having each thread print out data at certain times could help you identify the race condition.

Valgrind

Valgrind is a suite of tools designed to provide debugging and profiling tools to make your programs more correct and detect some runtime issues. The most used of these tools is Memcheck, which can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behaviour (for example, unfreed memory buffers). To run Valgrind on your program:

```
valgrind --leak-check=full --show-leak-kinds=all myprogram arg1 arg2
```

Arguments are optional and the default tool that will run is Memcheck. The output will be presented in form of number of allocations, number of freed allocations, and the number of errors.

Suppose we have a simple program like this:

```

#include <stdlib.h>

void dummy_function() {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;        // error 1: as you can see here we write to an out
                      // of bound memory address
}                    // error 2: memory leak the allocated x not freed

int main(void) {
    dummy_function();
    return 0;
}

```

This program compiles and runs with no errors. Let's see what Valgrind will output.

```

==29515== Memcheck, a memory error detector
==29515== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et
al.
==29515== Using Valgrind-3.11.0 and LibVEX; rerun with -h for
copyright info
==29515== Command: ./a
==29515==
==29515== Invalid write of size 4
==29515==   at 0x400544: dummy_function (in
/home/rafi/projects/exocpp/a)
==29515==   by 0x40055A: main (in /home/rafi/projects/exocpp/a)
==29515== Address 0x5203068 is 0 bytes after a block of size 40
alloc'd
==29515==   at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==29515==   by 0x400537: dummy_function (in
/home/rafi/projects/exocpp/a)
==29515==   by 0x40055A: main (in /home/rafi/projects/exocpp/a)
==29515==
==29515== HEAP SUMMARY:
==29515==   in use at exit: 40 bytes in 1 blocks
==29515== total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==29515==
==29515== LEAK SUMMARY:
==29515==   definitely lost: 40 bytes in 1 blocks
==29515==   indirectly lost: 0 bytes in 0 blocks
==29515==   possibly lost: 0 bytes in 0 blocks
==29515==   still reachable: 0 bytes in 0 blocks
==29515==     suppressed: 0 bytes in 0 blocks
==29515== Rerun with --leak-check=full to see details of leaked memory
==29515==
==29515== For counts of detected and suppressed errors, rerun with: -v
==29515== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from
0)

```

Invalid write: It detected our heap block overrun, writing outside of allocated block.

Definitely lost: Memory leak — you probably forgot to free a memory block.

Valgrind is a very effective tool to check for errors at runtime. C is very special when it comes to such behavior, so after compiling your program you can use Valgrind to fix errors that your compiler may not catch and that usually happen when your program is running.

For more information, you can refer to the valgrind website.

TSAN

ThreadSanitizer is a tool from Google, built into clang and gcc, to help you detect race conditions in your code. For more information about the tool, see the Github wiki. Note, that running with tsan will slow your code down a bit. Consider the following code.

```
#include <pthread.h>
#include <stdio.h>

int global;

void *Thread1(void *x) {
    global++;
    return NULL;
}

int main() {
    pthread_t t[2];
    pthread_create(&t[0], NULL, Thread1, NULL);
    global = 100;
    pthread_join(t[0], NULL);
}

// compile with gcc -fsanitize=thread -pie -fPIC -ltsan -g
// simple_race.c
```

We can see that there is a race condition on the variable `global`. Both the main thread and the thread created with `pthread_create` will try to change the value at the same time. But, does ThreadSanitizer catch it?

```

$ ./a.out
=====
WARNING: ThreadSanitizer: data race (pid=28888)
  Read of size 4 at 0x7f73ed91c078 by thread T1:
    #0 Thread1 /home/zmick2/simple_race.c:7 (exe+0x000000000a50)
    #1 :0 (libtsan.so.0+0x00000001b459)

  Previous write of size 4 at 0x7f73ed91c078 by main thread:
    #0 main /home/zmick2/simple_race.c:14 (exe+0x000000000ac8)

  Thread T1 (tid=28889, running) created by main thread at:
    #0 :0 (libtsan.so.0+0x00000001f6ab)
    #1 main /home/zmick2/simple_race.c:13 (exe+0x000000000ab8)

SUMMARY: ThreadSanitizer: data race /home/zmick2/simple_race.c:7
  Thread1
=====
ThreadSanitizer: reported 1 warnings

```

If we compiled with the debug flag, then it would give us the variable name as well.

GDB

Introduction to gdb

Setting breakpoints programmatically A very useful trick when debugging complex C programs with GDB is setting breakpoints in the source code.

```

int main() {
    int val = 1;
    val = 42;
    asm("int $3"); // set a breakpoint here
    val = 7;
}

```

```

$ gcc main.c -g -o main && ./main
(gdb) r
[...]
Program received signal SIGTRAP, Trace/breakpoint trap.
main () at main.c:6
6     val = 7;
(gdb) p val
$1 = 42

```

Checking memory content Memory Content

For example,

```
int main() {
    char bad_string[3] = {'C', 'a', 't'};
    printf("%s", bad_string);
}
```

```
$ gcc main.c -g -o main && ./main
$ Cat ZVQiFj- $
```

```
(gdb) l
1 #include <stdio.h>
2 int main() {
3     char bad_string[3] = {'C', 'a', 't'};
4     printf("%s", bad_string);
5 }
(gdb) b 4
Breakpoint 1 at 0x100000f57: file main.c, line 4.
(gdb) r
[...]
Breakpoint 1, main () at main.c:4
4     printf("%s", bad_string);
(gdb) x/16xb bad_string
0x7fff5fbff9cd: 0x63 0x61 0x74 0xe0 0xf9 0xbf 0x5f 0xff
0x7fff5fbff9d5: 0x7f 0x00 0x00 0xfd 0xb5 0x23 0x89 0xff
(gdb)
```

Here, by using the x command with parameters 16xb, we can see that starting at memory address 0x7fff5fbff9c (value of bad_string), printf would actually see the following sequence of bytes as a string because we provided a malformed string without a null terminator.

Life in the terminal

TODO

```
0x43 0x61 0x74 0xe0 0xf9 0xbf 0x5f 0xff 0x7f 0x00
```

System Programming Jokes

Warning: Authors are not responsible for any neuro-apoptosis caused by these “jokes.” - Groaners are allowed.

Light bulb jokes

Q. How many system programmers does it take to change a lightbulb?

A. Just one but they keep changing it until it returns zero.

A. None they prefer an empty socket.

A. Well you start with one but actually it waits for a child to do all of the work.

Groaners

Why did the baby system programmer like their new colorful blankie? It was multithreaded.

Why are your programs so fine and soft? I only use 400-thread-count or higher programs.

Where do bad student shell processes go when they die? Forking Hell.

Why are C programmers so messy? They store everything in one big heap.

System Programmer (Definition)

A system programmer is...

Someone who knows `sleepsort` is a bad idea but still dreams of an excuse to use it.

Someone who never lets their code deadlock... but when it does, causes more problems than everyone else combined.

Someone who believes zombies are real.

Someone who doesn't trust their process to run correctly without testing with the same data, kernel, compiler, RAM, filesystem size, file system format, disk brand, core count, CPU load, weather, magnetic flux, orientation, pixie dust, horoscope sign, wall color, wall gloss and reflectance, motherboard, vibration, illumination, backup battery, time of day, temperature, humidity, lunar position, sun-moon, co-position...

System Program (Definition)

A system program ...

Evolves until it can send email.

Evolves until it has the potential to create, connect and kill other programs and consume all possible CPU, memory, network, ... resources on all possible devices but chooses not to. Today.