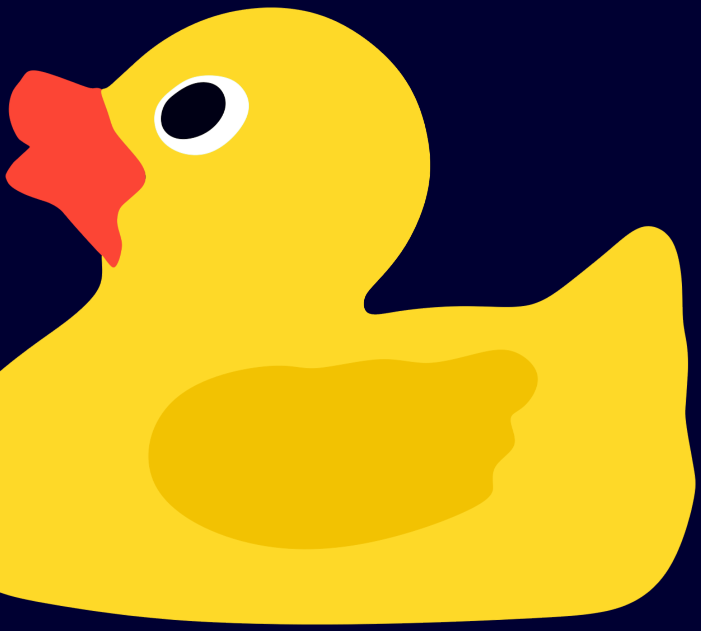
Coursebook



B. Venkatesh, L. Angrave, et Al.

Contents

List of Figures

List of Tables

1 Introduction

To thy happy children of the future, those of the past send greetings.

Alma Mater

At the University of Illinois at Urbana-Champaign, I fundamentally believe that we have a right to make the university better for all future students. It is a message etched into our Alma Mater and makes up the DNA of our course staff. As such, we created the coursebook. The coursebook is a free and open systems programming textbook that anyone can read, contribute to, and modify for now and forever. We don't think information should be behind a walled garden, and we truly believe that complex concepts can be explained simply and fully, for anyone to understand. The goal of this book is to teach you the basics and give you some intuition into the complexities of systems programming.

Like any good book, it isn't complete. We still have plenty of examples, ideas, typos, and chapters to work on. If you find any issues, please file an issue or email a list of typos to CS 241 course staff, and we'll be happy to work on it. We are constantly trying to make the book better for students a year and ten years from now.

This work is based on the original coursebook located at at this url. All these peoples' hard work is included in the section below.

Thanks again and happy reading!

– Bhuvy

Authors

```
Lawrence Angrave <angrave@illinois.edu>
Bhuvan Venkatesh <bhuvan.venkatesh21@gmail.com>
joebenassi <joebenassi@gmail.com>
jakebailey <zikaeroh@gmail.com>
Ebrahim Byagowi <ebrahim@gnu.org>
Alex Kizer <the.alex.kizer@gmail.com>
dimyr7 <dimyr7.puma@gmail.com>
Ed K <ed.karrels@gmail.com>
ace-n <nassri2@illinois.edu>
josephmilla <jjtmilla@gmail.com>
```

```
Thomas Liu <thomasliu02@gmail.com>
Johnny Chang <johnny@johnnychang.com>
goldcase <johnny@johnnychang.com>
vassimladenov <vassi1995@icloud.com>
SurtaiHan <surtai.han@gmail.com>
Brandon Chong <bchong95@users.noreply.github.com>
Ben Kurtovic <ben.kurtovic@gmail.com>
dprorok2 <dprorok2@illinois.edu>
anchal-agrawal <aagrawa4@illinois.edu>
Lawrence Angrave <angrave@illinois.eduuutoomanyu>
daeyun <daeyunshin@gmail.com>
bchong95 <bschong2@illinois.edu>
rushingseas8 <georgealeks@hotmail.com>
lukspdev <lllluuukke@gmail.com>
hilalh <habashi2@illinois.edu>
dimyr7 <dimyr7@hotmail.com>
Azrakal <genxswordsman@hotmail.com>
G. Carl Evans <gcevans@gmail.com>
Cornel Punga <cornel.punga@gmail.com>
vikasagartha <vikasagartha@gmail.com>
dyarbrough93 <dyarbrough93@yahoo.com>
berwin7996 <berwin7996@gmail.com>
Sudarshan Govindaprasad <SudarshanGp@users.noreply.github.com>
NMyren <ntmyren@gmail.com>
Ankit Gohel <ankitgohel1996@gmail.com>
vha-weh-shh <bhaweshchhetri1@gmail.com>
sasankc <sasank.chundi@gmail.com>
rishabhjain2795 <rishabhjain2795@gmail.com>
nickgarfield <nickgarfield@icloud.com>
by700git <aaabox@yeah.net>
bw-vbnm <bwang19@illinois.edu>
Navneeth Jayendran <jayndrn2@illinois.edu>
Joe Benassi <joebenassi@gmail.com>
Harpreet Singh <hshssingh4@gmail.com>
FenixFeather <thomasliu02@gmail.com>
EntangledLight <bdelapor@illinois.edu>
Bliss Chapman <bliss.chapman@gmail.com>
zikaeroh <zikaeroh@gmail.com>
time bandit <radicalrafi@gmail.com>
paultgibbons <paultgibbons@gmail.com>
kevinwang <kevin@kevinwang.com>
cPolaris <cPolaris@users.noreply.github.com>
Zecheng () <zzhan147@illinois.edu>
Wieschie <supernova190@gmail.com>
WeiL <z920631580@gmail.com>
Graham Dyer <gdyer2@illinois.edu>
Arun Prakash Jana <engineerarun@gmail.com>
Ankit Goel <ankitgoel616@gmail.com>
```

```
Allen Kleiner <akleiner24@gmail.com>
Abhishek Deep Nigam <adn5327@users.noreply.github.com>
zmmille2 <zmmille2@gmail.com>
sidewallme <sidewallme@gmail.com>
raych05 <raymondcheng05@gmail.com>
mmahes <malinixmahes@gmail.com>
mass <amass1212@gmail.com>
kovaka <jakelagrou@gmail.com>
gmag23 <gmag23@gmail.com>
ejian2 <ejian2@illinois.edu>
cerutii <marc.ceruti@gmail.com>
briantruong777 <briantruong777@gmail.com>
adevar <adevar2@illinois.edu>
Yuxuan Zou (Sean) <yzouac@connect.ust.hk>
Xikun Zhang <xikunz2@illinois.edu>
Vishal Disawar <disawar2@illinois.edu>
Taemin Shin <cprayer@naver.com>
Sujay Patwardhan <sujay.patwardhan@gmail.com>
SufeiZ <sufeizhang92@gmail.com>
Sufei Zhang <sufeizhang92@gmail.com>
Steven Shang <sstevenshang@users.noreply.github.com>
Steve Zhu <st.zhu1@gmail.com>
Sibo Wang <sibowsb@gmail.com>
Shane Ryan <shane1027@users.noreply.github.com>
Scott Bigelow <epheph@gmail.com>
Riyad Shauk <riyadshauk@users.noreply.github.com>
Nathan Somers <nsomers2@illinois.edu>
LieutenantChips <vkaraku2@illinois.edu>
Jacob K LaGrou <jakelagrou@gmail.com>
George <ruan3@illinois.edu>
David Levering <dmlevering@gmail.com>
Bernard Lim <bernlim93@users.noreply.github.com>
zwang180 <zshwang0809@gmail.com>
xuanwang91 <LilyBiology2010@gmail.com>
xin-0 <xintong2@illinois.edu>
wchill <wchill1337@gmail.com>
vishnui <vishnui@gmail.com>
tvarun2013 <tvarun2013@gmail.com>
sstevenshang <sstevenshang@users.noreply.github.com>
ssquirrel <lxl_zhang@Hotmail.com>
smeenai <shoaib.meenai@gmail.com>
shrujancheruku <shrujancheruku@gmail.com>
ruiqili2 <ruiqili2@users.noreply.github.com>
rchwlsk2 <rchwlsk2@illinois.edu>
ralphchung <ralphchung2005@gmail.com>
nikioftime <ncwells2@illinois.edu>
mosaic0123 <truffer@live.com>
majiasheng <jiasheng.ma@yahoo.com>
```

```
m <cheonghiuwaa@gmail.com>
li820970 <li820970@gmail.com>
kuck1 <kuck1@illinois.edu>
kkgomez2 <kkgomez2@users.noreply.github.com>
jjames34 <James_Jerry1@yahoo.com>
jargals2 <jargals2@ilinois.edu>
hzding621 <hzding621@users.noreply.github.com>
hzding621 <hzding621@gmail.com>
hsingh23 <hisingh1@gmail.com>
denisdemaisbr <denis@roo.com.br>
daishengliang <daishengliang@gmail.com>
cucumbur <bomblolism@gmail.com>
codechao999 <brianweis@comcast.net>
chrisshroba <chrisshroba@gmail.com>
cesarcastmore <cesar.cast.more@gmail.com>
briantruong777 <briantruong777@users.noreply.github.com>
botengHY <tengbo1992@gmail.com>
blapalp <pzkmmmh@gmail.com>
bchhetri1 <bhaweshchhetri1@gmail.com>
anadella96 <aisha.nadella@gmail.com>
akleiner2 <akleiner24@gmail.com>
aRatnam12 <ansh.ratnam@gmail.com>
Yash Sharma <yashosharma@gmail.com>
Xiangbin Hu <xhu27@illinois.edu>
WininWin <ezoneid@gmail.com>
William Klock <william.klock@gmail.com>
WenhanZ <marinebluee@hotmail.com>
Vivek Pandya <vivekvpandya@gmail.com>
Vineeth Puli <vpuli98@gmail.com>
Vangelis Tsiatsianas <vangelists@users.noreply.github.com>
Vadiml1024 <vadim@mbdsys.com>
Utsav2 <ukshah2@illinois.edu>
Thirumal Venkat <zapstar@users.noreply.github.com>
TheEntangledLight <bdelapor@illinois.edu>
SudarshanGp <SudarshanGp@users.noreply.github.com>
Sudarshan Konge <6025419+sudk1896@users.noreply.github.com>
Slix <slixpk@gmail.com>
Sasank Chundi <sasank.chundi@gmail.com>
SachinRaghunathan <srghnth2@illinois.edu>
Rémy Léone <remy.leone@gmail.com>
RusselLuo <russelluo@gmail.com>
Roman Vaivod <littlewhywhat@gmail.com>
Rohit Sarathy <rohit@sarathy.org>
Rick Sheahan <bomblolism@gmail.com>
Rakhim Davletkaliyev <freetonik@gmail.com>
Punitvara <punitvara@gmail.com>
Phillip Quy Le <pitlv2109@gmail.com>
Pavle Simonovic <simonov2@illinois.edu>
```

```
Paul Hindt <phindt@gmail.com>
Nishant Maniam <nishant.maniam@gmail.com>
Mustafa Altun <gmail@mustafaaltun.com>
Mohammed Sadik P. K <sadiqpkp@gmail.com>
Mingchao Zhang <43462732+mingchao-zhang@users.noreply.github.com>
Michael Vanderwater <vndrwtr2@users.noreply.github.com>
Maxiwell Luo <maxluoXIII@gmail.com>
LunaMystic <suxianghan@outlook.com>
Liam Monahan <liam@liammonahan.com>
Joshua Wertheim <joshwertheim@gmail.com>
John Pham <newhope11134@gmail.com>
Johannes Scheuermann <johscheuer@users.noreply.github.com>
Joey Bloom <15joeybloom@users.noreply.github.com>
Jimmy Zhang <midnight.vivian@gmail.com>
Jeffrey Foster <jmfoste2@illinois.edu>
James Daniel <james-daniel@users.noreply.github.com>
Jake Bailey <zikaeroh@gmail.com>
JACKHAHA363 <luyuchen.paul@gmail.com>
Hydrosis <badda2k@gmail.com>
Hong <plantvsbird@gmail.com>
Grant Wu <grantwu2@gmail.com>
EvanFabry <Evan.Fabry@gmail.com>
EddieVilla <EddieVilla@users.noreply.github.com>
Deepak Nagaraj <n.deepak@gmail.com>
Daniel Meir Doron <danielmeirdoron@gmail.com>
Daniel Le <GreenRecycleBin@gmail.com>
Daniel Jamrozik <djamro2@illinois.edu>
Daniel Carballal <danielenriquecarballal@gmail.com>
Daniel <DTV96Calibre@users.noreply.github.com>
Daeyun Shin <daeyun@daeyunshin.com>
Creyslz <creyslz@gmail.com>
Christian Cygnus <gamer00@att.net>
CharlieMartell <charliecmartell@gmail.com>
Caleb Bassi <calebjbassi@gmail.com>
Brian Kurek <brkurek@gmail.com>
Brendan Wilson <brendan.x.wilson@gmail.com>
Bo Liu <boliu1@illinois.edu>
Ayush Ranjan <ayushr2@illinois.edu>
Atul kumar Agrawal <ms.atul1303@gmail.com>
Artur Sak <artursak1981@gmail.com>
Ankush Agarwal <ankushagarwal@users.noreply.github.com>
Angelino <angelino_m@outlook.com>
Andrey Zaytsev <andzaytsev@gmail.com>
Alex Yang <alyx.yang@gmail.com>
Alex Cusack <cusackalex@gmail.com>
Aidan Epstein <aidan@jmad.org>
Ace Nassri <ace.nassri@gmail.com>
Abdullahi Abdalla <abdalla6@illinois.edu>
```

```
Aneesh Durg <durg2@illinois.edu>
Assassin Eclipse <hungwoei96@hotmail.com>
Eric Cao <eric7252000@gmail.com>
Raphael Long <rafilong42@gmail.com>
WeiL <z920631580@gmail.com>
williamsentosa95
    <38774380+williamsentosa95@users.noreply.github.com>
Pradyumna Shome <pradyumna.shome@gmail.com>
Benjamin West Pollak <benjaminwpollak@gmail.com>
```

2

Background

> Sometimes the journey of a thousand steps begins by learning to walk
>
> Bhuvy

Systems Architecture

This section is a short review of System Architecture topics that you'll need for System Programming.

Assembly

What is assembly? Assembly is the lowest that you'll get to machine language without writing 1's and 0's. Each computer has an architecture, and that architecture has an associated assembly language. Each assembly command has a 1:1 mapping to a set of 1's and 0's that tell the computer exactly what to do. For example, the following in the widely used x86 Assembly language add one to the memory address 20 [13] – you can also look in [8] Section 2A under the add instruction though it is more verbose.

```
add BYTE PTR [0x20], 1
```

Why do we mention this? Because it is important that although you are going to be doing most of this class in C. That this is what the code is translated into. Serious implications arise for race conditions and atomic operations.

Atomic Operations

An operation is atomic if no other processor should interrupt it. Take for example the above assembly code to add one to a register. In the architecture, it may actually have a few different steps on the circuit. The operation may start by fetching the value of the memory from the stick of ram, then storing it in the cache or a register, and then finally writing back [12] – under the description for *fetch-and-add* though your micro-architecture may vary. Or depending on performance operations, it may keep that value in cache or in a register which is local to that process – try dumping the `-O2` optimized assembly of incrementing a variable. The problem comes in if two processors try to do it at the same time. The two processors could at the same time copy the value of the memory address, add one, and store the same result back, resulting in the value only being incremented once. That is why we have a special set of instructions on modern systems called atomic operations. If an instruction is atomic, it makes sure that only one processor or thread performs any intermediate step at a time. With x86 this is done by the `lock` prefix [8, p. 1120].

```
lock add BYTE PTR [0x20], 1
```

Why don't we do this for everything? It makes commands slower! If every time a computer does something it has to make sure that the other cores or processors aren't doing anything, it'll be much slower. Most of the time we differentiate these with special consideration. Meaning, we will tell you when we use something like this. Most of the time you can assume the instructions are unlocked.

Caching

Ah yes, Caching. One of computer science's greatest problems. Caching that we are referring is processor caching. If a particular address is already in the cache when reading or writing, the processor will perform the operation on the cache such as adding and update the actual memory later because updating memory is slow [9, Section 3.4]. If it isn't, the processor requests a chunk of memory from the memory chip and stores it in the cache, kicking out the least recently used page – this depends on caching policy, but Intel's does use this. This is done because the l3 processor cache is roughly three times faster to reach than the memory in terms of time [11, p. 22] though exact speeds will vary based on the clock speed and architecture. Naturally, this leads to problems because there are two different copies of the same value, in the cited paper this refers to an unshared line. This isn't a class about caching, know how this could impact your code. A short but non-complete list could be

1. Race Conditions! If a value is stored in two different processor caches, then that value should be accessed by a single thread.

2. Speed. With a cache, your program may look faster mysteriously. Just assume that reads and writes that either happened recently or are next to each other in memory are fast.

3. Side effects. Every read or write effects the cache state. While most of the time this doesn't help or hurt, it is important to know. Check the Intel programmer guide on the lock prefix for more information.

Interrupts

Interrupts are a important part of system programming. An interrupt is internally an electrical signal that is delivered to the processor when something happens – this is a hardware interrupt [3]. Then the hardware decides if this is something that it should handle (i.e. handling keyboard or mouse input for older keyboard and mouses) or it should pass to the operating system. The operating system then decides if this is something that it should handle (i.e. paging a memory table from disk) or something the application should handle (i.e. a

SEGFAULT). If the operating system decides that this is something that the process or program should take care of, it sends a **software fault** and that software fault is then propagated. The application then decides if it is an error (SEGFAULT) or not (SIGPIPE for example) and reports to the user. Applications can also send signals to the kernel and to the hardware as well. This is an oversimplification because there are certain hardware faults that can't be ignored or masked away, but this class isn't about teaching you to build an operating system.

An important application of this is this is how system calls are served! There is a well-established set of registers that the arguments go in according to the kernel as well as a system call "number" again defined by the kernel. Then the operating system triggers an interrupt which the kernel catches and serves the system call [7].

Operating system developers and instruction set developers alike didn't like the overhead of causing an interrupt on a system call. Now, systems use `SYSENTER` and `SYSEXIT` which has a cleaner way of transferring control safely to the kernel and safely back. What safely means is obvious out of the scope for this class, but it persists.

Optional: Hyperthreading

Hyperthreading is a new technology and is in no way shape or form multithreading. Hyperthreading allows one physical core to appear as many virtual cores to the operating system [8, P.51]. The operating system can then schedule processes on these virtual cores and one core will execute them. Each core interleaves processes or threads. While the core is waiting for one memory access to complete, it may perform a few instructions of another process thread. The overall result is more instructions executed in a shorter time. This potentially means that you can divide the number of cores you need to power smaller devices.

There be dragons here though. With hyperthreading, you must be wary of optimizations. A famous hyperthreading bug that caused programs to crash if at least two processes were scheduled on a physical core, using specific registers, in a tight loop. The actual problem is better explained through an architecture lens. But, the actual application was found through systems programmers working on OCaml's mainline [10].

Debugging and Environments

I'm going to tell you a secret about this course: it is about working smarter *not* harder. The course can be time-consuming but the reason that so many people see it as such (and why so many students don't see it as such) is the relative familiarity of people with their tools. Let's go through some of the common tools that you'll be working on and need to be familiar with.

ssh

`ssh` is short for the Secure Shell [2]. It is a network protocol that allows you to spawn a shell on a remote machine. Most of the times in this class you will need to ssh into your VM like this

```
$ ssh netid@sem-cs241-VM.cs.illinois.edu
```

If you don't want to type your password out every time, you can generate an ssh key that uniquely identifies your machine. If you already have a key pair, you can skip to the copy id stage.

```
> ssh-keygen -t rsa -b 4096
# Do whatever keygen tells you
# Don't feel like you need a passcode if your login password is
    secure
> ssh-copy-id netid@sem-cs241-VM.cs.illinois.edu
# Enter your password for maybe the final time
> ssh netid@sem-cs241-VM.cs.illinois.edu
```

If you still think that that is too much typing, you can always alias hosts. You may need to restart your VM or reload sshd for this to take effect. The config file is available on Linux and Mac distros. For Windows, you'll have to use the Windows Linux Subsystem or configure any aliases in PuTTY

```
> cat ~/.ssh/config
Host vm
  User          netid
  HostName      sem-cs241-VM.cs.illinois.edu
> ssh vm
```

git

What is ‘git‘? Git is a version control system. What that means is git stores the entire history of a directory. We refer to the directory as a repository. So what do you need to know is a few things. First, create your repository with the repo creator. If you haven't already signed into enterprise GitHub, make sure to do so otherwise your repository won't be created for you. After that, that means your repository is created on the server. Git is a decentralized version control system, meaning that you'll need to get a repository onto your VM. We can do this with a clone. Whatever you do, **do not go through the README.md tutorial**.

```
$ git clone
    https://github-dev.cs.illinois.edu/cs241-fa18/<netid>.git
```

This will create a local repository. The workflow is you make a change on your local repository, add the changes to a current commit, actually commit, and push the changes to the server.

```
$ # edit the file, maybe using vim
$ git add <file>
$ git commit -m "Committing my file"
$ git push origin master
```

Now to explain git well, you need to understand that git for our purposes will look like a linked list. You will always be at the head of master, and you will do the edit-add-commit-push loop. We have a separate branch on Github that we push feedback to under a specific branch which you can view on Github website. The markdown file will have information on the test cases and results (like standard out).

Every so often git can break. Here is a list of commands you probably won't need to fix your repo

1. git-cherry-pick
2. git-pack
3. git-gc
4. git-clean
5. git-rebase
6. git-stash/git-apply/git-pop
7. git-branch

If you are currently on a branch, and you don't see either

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

or

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
      directory)

        modified:  <FILE>
        ...

no changes added to commit (use "git add" and/or "git commit -a")
```

And something like

```
$ git status
HEAD detached at 4bc4426
nothing to commit, working directory clean
```

Don't panic, but your repository may be in an unworkable state. If you aren't nearing a deadline, come to office hours or ask your question on Piazza, and we'd be happy to help. In an emergency scenario, delete your repository and re-clone (you'll have to add the `release` as above). **This will lose any local uncommitted changes. Make sure to copy any files you were working on outside the directory, remove and copy them back in**

If you want to learn more about git, there are all but an endless number of tutorials and resources online that can help you. Here are some links that can help you out

1. `https://git-scm.com/docs/gittutorial`
2. `https://www.atlassian.com/git/tutorials/what-is-version-control`
3. `https://thenewstack.io/tutorial-git-for-absolutely-everyone/`

Editors

Some people take this as an opportunity to learn a new editor, others not so much. The first part is those of you who want to learn a new editor. In the editor war that spans decades, we have come to the battle of vim vs emacs.

Vim is a text editor and a Unix-like utility. You enter vim by typing `vim [file]`. This takes you into the editor. You start off in normal mode. In this mode, you can move around with many keys with the most common ones being `jklh`. To exit vim from this mode, you need to type `:q` which quits. If you have any unsaved edits, you must either save them `:w`, save and quit `:wq`, or discard changes `:q!`. To make edits you can either type `i` to change you into insert mode or `a` to change to insert mode **a**fter the cursor. This is the basics when it comes to vim

Emacs is more of a way of life, and I don't mean that figuratively. A lot of people say that emacs is a powerful operating system lacking a decent text editor. This means emacs can house a terminal, gdb session, ssh session, code and a whole lot more. It would not be fitting any other way to introduce you to the gnu-emacs any other way than the gnu-docs `https://www.gnu.org/software/emacs/tour/`. Just note that emacs is *insanely* powerful. You can do almost anything with it. There are a fair number of students who like the IDE-aspect of other programming languages. Know that you can set up emacs to be an IDE, but you have to learn a bit of Lisp `http://martinsosic.com/development/emacs/2017/12/09/emacs-cpp-ide.html`.

Then there are those of you who like to use your own editors. That is completely fine. For this, we require sshfs which has ports on many different machines.

1. Windows `https://github.com/billziss-gh/sshfs-win`
2. Mac `https://github.com/osxfuse/osxfuse/wiki/SSHFS`
3. Linux `https://help.ubuntu.com/community/SSHFS`

At that point, the files on your VM are synced with the files on your machine and edits can be made and will be synced.

At the time of writing, Bhuvy (That's me!) likes to use spacemacs `http://spacemacs.org/` which marries both vim and emacs and both of their difficulties. I'll give my soapbox for why I like it, but be warned that if you are starting from absolutely no vim or emacs experience the learning curve along with this course may be too much.

1. Extensible. Spacemacs has a clean design written in lisp. There are 100s of packages ready to be installed by editing your spacemacs config and reloading that do everything from syntax checking, automatic static analyzing, etc.
2. Most of the good parts from vim and emacs. Emacs is good at doing everything by being a fast editor. Vim is good at making fast edits and moving around. Spacemacs is the best of both worlds allowing vim keybindings to all the emacs goodness underneath.

3. Lots of preconfiguration done. As opposed with a fresh emacs install, a lot of the configurations with language and projects are done for you like neotree, helm, various language layers. All that you have to do is navigate neotree to the base of your project and emacs will turn into an IDE for that programming language.

But obviously to each his or her own. Many people will argue that editor gurus spend more time editing their editors and actually editing.

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 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,

```c
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

```c
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, this class is about writing system programs, not a class about refactoring/debugging your code. In fact, most kernel code is so atrocious that you don't want to read it – the defense there is that it needs to be. 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 == node->next->prev) 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, non-null, ->size is reasonable, etc. The DEBUG macro will disable all assertions, so don't forget to set that once you finish debugging [1].

Here is a quick example with an assert. Let's say that we are writing code using memcpy. We would want to put an assert before that checks whethermy two memory regions overlap. If they do overlap, memcpy runs into undefined behavior, so we want to catch that problem than later.

```
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!

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 [4]. 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 behavior (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 the form: the number of allocations, frees, and 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: Out of bounds write, as you can see
        here we write to an out of bound memory address.
}                    // error 2: Memory Leak, x is allocated at
    function exit.

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

Valgrind is a effective tool to check for errors at runtime. C is special when it comes to such behavior, so after compiling your program you can use Valgrind to fix errors that your compiler may miss and that usually happens when your program is running.

For more information, you can refer to the manual [4]

TSAN

ThreadSanitizer is a tool from Google, built into clang and gcc, to help you detect race conditions in your code [5]. Note, that running with tsan will slow your code down a bit. Consider the following code.

```c
#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 created thread will try to change the value at the same time. But, does ThreadSantizer 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

GDB is short for the GNU Debugger. GDB is a program that helps you track down errors by interactively debugging them [6]. It can start and stop your program, look around, and put in ad hoc constraints and checks. Here are a few examples.

Setting breakpoints programmatically A breakpoint is a line of code where you want the execution to stop and give control back to the debugger. A useful trick when debugging complex C programs with GDB is setting breakpoints in the source code.

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

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

You can also set breakpoints programmatically. Assume that we have no optimization and the line numbers are as follows

```
1. int main() {
```

```
2.     int val = 1;
3.     val = 42;
4.     val = 7;
5. }
```

We can now set the breakpoint before the program starts.

```
$ gcc main.c -g -o main
$ gdb --args ./main
(gdb) break main.c:4
[...]
(gdb) p val
$1 = 42
```

Checking memory content We can also use gdb to check the content of different pieces of memory. For example,

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

Compiled we get

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

We can now use gdb to look at specific bytes of the string and reason about when the program should've stopped running

```
(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.

Involved gdb example

Here is how one of your TAs would go through and debug a simple program that is going wrong. First, the source code of the program. If you can see the error immediately, please bear with us.

```
#include <stdio.h>

double convert_to_radians(int deg);

int main(){
    for (int deg = 0; deg > 360; ++deg){
        double radians = convert_to_radians(deg);
        printf("%d. %f\n", deg, radians);
    }
    return 0;
}

double convert_to_radians(int deg){
    return ( 31415 / 1000 ) * deg / 180;
}
```

How can we use gdb to debug? First we ought to load GDB.

```
$ gdb --args ./main
(gdb) layout src; # If you want a GUI type
(gdb) run
(gdb)
```

Want to take a look at the source?

```
(gdb) l
1   #include <stdio.h>
2
3   double convert_to_radians(int deg);
4
5   int main(){
6       for (int deg = 0; deg > 360; ++deg){
7           double radians = convert_to_radians(deg);
8           printf("%d. %f\n", deg, radians);
9       }
10      return 0;
(gdb) break 7 # break <file>:line or break <file>:function
(gdb) run
(gdb)
```

From running the code, the breakpoint didn't even trigger, meaning the code never got to that point. Thats because of the comparison! Okay, flip the sign it should work now right?

```
(gdb) run
350. 60.000000
351. 60.000000
352. 60.000000
353. 60.000000
354. 60.000000
355. 61.000000
356. 61.000000
357. 61.000000
358. 61.000000
359. 61.000000
```

```
(gdb) break 14 if deg == 359 # Let's check the last iteration only
(gdb) run
...
(gdb) print/x deg # print the hex value of degree
$1 = 0x167
(gdb) print (31415/1000)
$2 = 0x31
(gdb) print (31415/1000.0)
$3 = 201.749
(gdb) print (31415.0/10000.0)
$4 = 3.1414999999999999
```

That was only the bare minimum, though most of you will get by with that. There are a whole load more resources on the web, here are a few specific ones that can help you get started.

1. Introduction to gdb

2. Memory Content

3. CppCon 2015: Greg Law Give me 15 minutes I'll change your view of GDB

Shell

What do you actually use to run your program? A shell! A shell is a programming language that is running inside your terminal. A terminal is merely a window to input commands. Now, on POSIX we usually have one shell called `sh` that is linked to a POSIX compliant shell called `dash`. Most of the time, you use a shell called `bash` that is somewhat POSIX compliant but has some nifty built-in features. If you want to be even more advanced, `zsh` has some more powerful features like tab complete on programs and fuzzy patterns.

Undefined Behavior Sanitizer

The undefined behavior sanitizer is a wonderful tool provided by the llvm project. It allows you to compile code with a runtime checker to make sure that you don't do undefined behavior for various categories. We will try to include it into our projects, but requires support form all the external libraries that we use so we may not get around to all of them. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

Undefined behavior - why we can't solve it in general

Also please please read Chris Lattner's 3 Part blog post on undefined behavior. It can shed light on debug builds and the mystery of compiler optimization.
`http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html`

Clang Static Build Tools

Clang provides a great drop-in replacement tools for compiling programs. If you want to see if there is an error that may cause a race condition, casting error, etc, all you need to do is the following.

```
$ scan-build make
```

And in addition to the make output, you will get static build warnings.

strace and ltrace

strace and ltrace are two programs that trace the system calls and library calls respectively of a running program or command. These may be missing on your system so to install feel free to run the following.

```
$ sudo apt install strace ltrace
```

Debugging with ltrace can be as simple as figuring out what was the return call of the last library call that failed.

```
int main() {
    FILE *fp = fopen("I don't exist", "r");
    fprintf(fp, "a");
    fclose(fp);
    return 0;
}
```

```
> ltrace ./a.out
 __libc_start_main(0x8048454, 1, 0xbfc19db4, 0x80484c0, 0x8048530
    <unfinished ...>
 fopen("I don't exist", "r")                          = 0x0
 fwrite("Invalid Write\n", 1, 14, 0x0 <unfinished ...>
 --- SIGSEGV (Segmentation fault) ---
 +++ killed by SIGSEGV +++
```

ltrace output can clue you in to weird things your program is doing live. Unfortunately, ltrace cant be used to inject faults, meaning that ltrace can tell you what is happening, but it can't tamper with what is already happening.

strace on the other hand could modify your program. Debugging with strace is amazing. The basic usage is running strace with a program, and itll get you a complete list of system call parameters.

```
$ strace head README.md
execve("/usr/bin/head", ["head", "README.md"], 0x7ffff28c8fa8 /*
    60 vars */) = 0
brk(NULL)                               = 0x7ffff5719000
access("/etc/ld.so.nohwcap", F_OK)  = -1 ENOENT (No such file or
    directory)
access("/etc/ld.so.preload", R_OK)  = -1 ENOENT (No such file or
    directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=32804, ...}) = 0
...
```

If the output is too verbose, you can use trace= with a command delimited list of syscalls to filter all but those calls.

```
$ strace -e trace=read,write head README.md
read(3,
    "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"...,
    832) = 832
read(3, "# Locale name alias data base.\n#"..., 4096) = 2995
read(3, "", 4096)                       = 0
read(3, "# C Datastructures\n\n[![Build Sta"..., 8192) = 1250
write(1, "# C Datastructures\n", 19# C Datastructures
```

You can also trace files or targets.

```
$ strace -e trace=read,write -P README.md head README.md
strace: Requested path 'README.md' resolved into
    '/mnt/c/Users/bhuvy/personal/libds/README.md'
read(3, "# C Datastructures\n\n[![Build Sta"..., 8192) = 1250
```

Newer versions of strace can actually inject faults into your program. This is useful when you want to occasionally make reads and writes fail for example in a networking application, which your program should handle. The problem is as of early 2019, that version is missing from Ubuntu repositories. Meaning that you'll have to install it from the source.

printfs

When all else fails, print! Each of your functions should have an idea of what it is going to do. 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.

To make printfs useful, try to have a macro that fills in the context by which the printf was called – a log statement if you will. A simple useful but untested log statement could be as follows. Try to make a test and figure out something that is going wrong, then log the state of your variables.

```
  #include <execinfo.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <stdarg.h>
  #include <unistd.h>

  // bt is print backtrace
  const int num_stack = 10;
```

```
int __log(int line, const char *file, int bt, const char *fmt,
    ...) {
  if (bt) {
    void *raw_trace[num_stack];
    size_t size = backtrace(raw_trace, sizeof(raw_trace) /
        sizeof(raw_trace[0]));
    char **syms = backtrace_symbols(raw_trace, size);

    for(ssize_t i = 0; i < size; i++) {
      fprintf(stderr, "|%s:%d| %s\n", file, line, syms[i]);
    }
    free(syms);
  }
  int ret = fprintf(stderr, "|%s:%d| ", file, line);
  va_list args;
  va_start(args, fmt);
  ret += vfprintf(stderr, fmt, args);
  va_end(args);
  ret += fprintf(stderr, "\n");
  return ret;
}

#ifdef DEBUG
#define log(...) __log(__LINE__, __FILE__, 0, __VA_ARGS__)
#define bt(...) __log(__LINE__, __FILE__, 1, __VA_ARGS__)
#else
#define log(...)
#define bt(...)
#endif

//Use as log(args like printf) or bt(args like printf) to either
    log or get backtrace

int main() {
  log("Hello Log");
  bt("Hello Backtrace");
}
```

And then use as appropriately.

Extra: Compiling and Linking

This is a high-level overview from the time you compile your program to the time you run your program. We often know that compiling your program is easy. You run the program through an IDE or a terminal, and it just works.

```
$ cat main.c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
$ gcc main.c -o main
$ ./main
Hello World!
$
```

Here are the rough stages of compiling for gcc.

1. Preprocessing: The preprocessor expands all preprocesor directives.
2. Parsing: The compiler parses the text file for function declarations, variable declarations, etc.
3. Assembly Generation: The compiler then generates assembly code for all the functions after some optimizations if enabled.
4. Assembling: The assembler turns the assembly into 0s and 1s and creates an object file. This object file maps names to pieces of code.
5. Static Linking: The linker then takes a series of objects and static libraries and resolves references of variables and functions from one object file to another. The linker then finds the main method and makes that the entry point for the function. The linker also notices when a function is meant to be dynamically linked. The compiler also creates a section in the executable that tells the operating system that these functions need addresses right before running.
6. Dynamic Linking: As the program is getting ready to be executed, the operating system looks at what libraries that the program needs and links those functions to the dynamic library.
7. The program is run.

Further classes will teach you about parsing and assembly – preprocessing is an extension of parsing. Most classes won't teach you about the two different types of linking though. Static linking a library is similar to combining object files. To create a static library, a compiler combines different object files to create one executable. A static library is literally is an archive of object files. These libraries are useful when you want your executable to be secure, you know all the code that is being included into your executable, and portable, all the code is bundled with your executable meaning no additional installs.

The other type is a dynamic library. Typically, dynamic libraries are installed user-wide or system-wide and are accessible by most programs. Dynamic libraries' functions are filled in right before they are run. There are a number of benefits to this.

- Lower code footprint for common libraries like the C standard library
- Late binding means more generalized code and less reliance on specific behavior.
- Differentiation means that the shared library can be updated while keeping the executable the same.

There are a number of drawbacks as well.

- All the code is no longer bundled into your program. This means that users have to install something else.
- There could be security flaws in the other code leading to security exploits in your program.
- Standard Linux allows you to "replace" dynamic libraries, leading to possible social engineering attacks.
- This adds additional complexity to your application. Two identical binaries with different shared libraries could lead to different results.

Homework 0

```
// First, can you guess which lyrics have been transformed into
   this C-like system code?
char q[] = "Do you wanna build a C99 program?";
#define or "go debugging with gdb?"
static unsigned int i = sizeof(or) != strlen(or);
char* ptr = "lathe";
size_t come = fprintf(stdout,"%s door", ptr+2);
int away = ! (int) * "";

int* shared = mmap(NULL, sizeof(int*), PROT_READ | PROT_WRITE,
   MAP_SHARED | MAP_ANONYMOUS, -1, 0);
munmap(shared,sizeof(int*));

if(!fork()) {
   execlp("man","man","-3","ftell", (char*)0); perror("failed");
}

if(!fork()) {
   execlp("make","make", "snowman", (char*)0);
      execlp("make","make", (char*)0));
}

exit(0);
```

So you want to master System Programming? And get a better grade than B?

m

```
int main(int argc, char** argv) {
   puts("Great! We have plenty of useful resources for you, but
      it's up to you to");
   puts(" be an active learner and learn how to solve problems and
      debug code.");
   puts("Bring your near-completed answers the problems below");
   puts(" to the first lab to show that you've been working on
      this.");
   printf("A few \"don't knows\" or \"unsure\" is fine for lab
      1.\n");
```

```
    puts("Warning: you and your peers will work hard in this
       class.");
    puts("This is not CS225; you will be pushed much harder to");
    puts(" work things out on your own.");
    fprintf(stdout,"This homework is a stepping stone to all future
       assignments.\n");
    char p[] = "So, you will want to clear up any confusions or
       misconceptions.\n";
    write(1, p, strlen(p) );
    char buffer[1024];
    sprintf(buffer,"For grading purposes, this homework 0 will be
       graded as part of your lab %d work.\n", 1);
    write(1, buffer, strlen(buffer));
    printf("Press Return to continue\n");
    read(0, buffer, sizeof(buffer));
    return 0;
}
```

Watch the videos and write up your answers to the following questions

Important!

The virtual machine-in-your-browser and the videos you need for HW0 are here:
`http://cs-education.github.io/sys/`

Questions? Comments? Use the current semester's CS241 Piazza: `https://piazza.com/`

The in-browser virtual machine runs entirely in JavaScript and is fastest in Chrome. Note the VM and any code you write is reset when you reload the page, **so copy your code to a separate document.** The post-video challenges are not part of homework 0, but you learn the most by doing rather than passively watching. You have some fun with each end-of-video challenge.

HW0 questions are below. Copy your answers into a text document because you'll need to submit them later in the course.

Chapter 1

In which our intrepid hero battles standard out, standard error, file descriptors and writing to files

1. **Hello, World! (system call style)** Write a program that uses `write()` to print out "Hi! My name is <Your Name>".

2. **Hello, Standard Error Stream!** Write a function to print out a triangle of height `n` to standard error. Your function should have the signature `void write_triangle(int n)` and should use `write()`. The triangle should look like this, for n = 3:

```
*
**
***
```

3. **Writing to files** Take your program from "Hello, World!" modify it write to a file called `hello_world.txt`. Make sure to use correct flags and a correct mode for `open()` (`man 2 open` is your friend).

4. **Not everything is a system call** Take your program from "Writing to files" and replace `write()` with `printf()`. *Make sure to print to the file instead of standard out!*

5. What are some differences between `write()` and `printf()`?

Chapter 2

Sizing up C types and their limits, `int` and `char` arrays, and incrementing pointers

1. How many bits are there in a byte?

2. How many bytes are there in a `char`?

3. How many bytes the following are on your machine? `int`, `double`, `float`, `long`, and `long long`

4. On a machine with 8 byte integers, the declaration for the variable `data` is `int data[8]`. If the address of data is `0x7fbd9d40`, then what is the address of `data+2`?

5. What is `data[3]` equivalent to in C? Hint: what does C convert `data[3]` to before dereferencing the address? Remember, the type of a string constant `"abc"` is an array.

6. Why does this SEGFAULT?

```c
char *ptr = "hello";
*ptr = 'J';
```

7. What is the value of the variable `str_size`?

```c
ssize_t str_size = sizeof("Hello\0World")
```

8. What is the value of the variable `str_len`

```c
ssize_t str_len = strlen("Hello\0World")
```

9. Give an example of X such that `sizeof(X)` is 3.

10. Give an example of Y such that `sizeof(Y)` might be 4 or 8 depending on the machine.

Chapter 3

Program arguments, environment variables, and working with character arrays (strings)

1. What are at least two ways to find the length of argv?
2. What does argv[0] represent?
3. Where are the pointers to environment variables stored (on the stack, the heap, somewhere else)?
4. On a machine where pointers are 8 bytes, and with the following code:

    ```
    char *ptr = "Hello";
    char array[] = "Hello";
    ```

    What are the values of sizeof(ptr) and sizeof(array)? Why?
5. What data structure manages the lifetime of automatic variables?

Chapter 4

Heap and stack memory, and working with structs

1. If I want to use data after the lifetime of the function it was created in ends, where should I put it? How do I put it there?
2. What are the differences between heap and stack memory?
3. Are there other kinds of memory in a process?
4. Fill in the blank: "In a good C program, for every malloc, there is a ___".
5. What is one reason malloc can fail?
6. What are some differences between time() and ctime()?
7. What is wrong with this code snippet?

    ```
    free(ptr);
    free(ptr);
    ```

8. What is wrong with this code snippet?

    ```
    free(ptr);
    printf("%s\n", ptr);
    ```

9. How can one avoid the previous two mistakes?

10. Create a `struct` that represents a `Person`. Then make a `typedef`, so that `struct Person` can be replaced with a single word. A person should contain the following information: their name (a string), their age (an integer), and a list of their friends (stored as a pointer to an array of pointers to `Person`s).

11. Now, make two persons on the heap, "Agent Smith" and "Sonny Moore", who are 128 and 256 years old respectively and are friends with each other. Create functions to create and destroy a Person (Person's and their names should live on the heap).

12. `create()` should take a name and age. The name should be copied onto the heap. Use malloc to reserve sufficient memory for everyone having up to ten friends. Be sure initialize all fields (why?).

13. `destroy()` should free up both the memory of the person struct and all of its attributes that are stored on the heap. Destroying one person keeps other people in tact any other.

Chapter 5

Text input and output and parsing using `getchar`, `gets`, and `getline`.

1. What functions can be used for getting characters from `stdin` and writing them to `stdout`?

2. Name one issue with `gets()`.

3. Write code that parses the string "Hello 5 World" and initializes 3 variables to "Hello", 5, and "World".

4. What does one need to define before including `getline()`?

5. Write a C program to print out the content of a file line-by-line using `getline()`.

C Development

These are general tips for compiling and developing using a compiler and git. Some web searches will be useful here

1. What compiler flag is used to generate a debug build?

2. You fix a problem in the Makefile and type `make` again. Explain why this may be insufficient to generate a new build.

3. Are tabs or spaces used to indent the commands after the rule in a Makefile?

4. What does `git commit` do? What's a `sha` in the context of git?

5. What does `git log` show you?

6. What does `git status` tell you and how would the contents of `.gitignore` change its output?

7. What does `git push` do? Why is it insufficient to commit with `git commit -m 'fixed all bugs'` ?

8. What does a non-fast-forward error `git push` reject mean? What is the most common way of dealing with this?

Optional: Just for fun

- Convert a song lyrics into System Programming and C code covered in this wiki book and share on Piazza.
- Find, in your opinion, the best and worst C code on the web and post the link to Piazza.
- Write a short C program with a deliberate subtle C bug and post it on Piazza to see if others can spot your bug.
- Do you have any cool/disastrous system programming bugs you've heard about? Feel free to share with your peers and the course staff on Piazza.

UIUC Specific Guidelines

Piazza

TAs and student assistants get a ton of questions. Some are well-researched, and some are not. This is a handy guide that'll help you move away from the latter and towards the former. Oh, and did I mention that this is an easy way to score points with your internship managers? Ask yourself...

1. Am I running on my Virtual Machine?
2. **Did I check the man pages?**
3. Have I searched for similar questions/followups on Piazza?
4. Have I read the MP/Lab specification completely?
5. Have I watched all of the videos?
6. Did I Google the error message and a few permutations thereof if necessary? How about StackOverflow.
7. Did I try commenting out, printing, and/or stepping through parts of the code bit by bit to find out precisely where the error occurs?
8. **Did I commit my code to git in case the TAs need more context?**
9. Did I include the console/GDB/Valgrind output **AND** code surrounding the bug in my Piazza post?
10. Have I fixed other segmentation faults unrelated to the issue I'm having?
11. Am I following good programming practice? (i.e. encapsulation, functions to limit repetition, etc)

The biggest tip that we can give you when asking a question on piazza if you want a swift answer is to **ask your question like you were trying to answer it**. Like before you ask a question, try to answer it yourself. If you are thinking about posting

> Hi, My code got a 50

Sounds good and courteous, but course staff would much much prefer a post resembling the following

> Hi, I recently failed test X, Y, Z which is about half the tests on this current assignment. I noticed that they all have something to do with networking and epoll, but couldn't figure out what was linking them together, or I may be completely off track. So to test my idea, I tried spawning 1000 clients with various get and put requests and verifying the files matched their originals. I couldn't get it to fail while running normally, the debug build, or valgrind or tsan. I have no warnings and none of the pre-syntax checks showed me anything. Could you tell me if my understanding of the failure is correct and what I could do to modify my tests to better reflect X, Y, Z? netid: bvenkat2

You don't need to be as courteous, though we'd appreciate it, this will get a faster response time hand over foot. If you were trying to answer this question, you'd have everything you need in the question body.

Bibliography

[1] assert. URL `http://www.cplusplus.com/reference/cassert/assert/`.

[2] ssh(1). URL `https://man.openbsd.org/ssh.1`.

[3] Chapter 3. hardware interrupts. URL `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/1.3/html/Realtime_Reference_Guide/chap-Realtime_Reference_Guide-Hardware_interrupts.html`.

[4] 4. memcheck: a memory error detector. URL `http://valgrind.org/docs/manual/mc-manual.html`.

[5] Threadsanitizercppmanual, Dec 2018. URL `https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual`.

[6] Gdb: The gnu project debugger, Feb 2019. URL `https://www.gnu.org/software/gdb/`.

[7] Manu Garg. Sysenter based system call mechanism in linux 2.6, 2006. URL `http://articles.manugarg.com/systemcallinlinux2_6.html`.

[8] Part Guide. Intel® 64 and ia-32 architectures software developers manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[9] CAT Intel. Improving real-time performance by utilizing cache allocation technology. *Intel Corporation, April*, 2015.

[10] Xavier Leroy. How i found a bug in intel skylake processors, Jul 2017. URL `http://gallium.inria.fr/blog/intel-skylake-bug/`.

[11] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30:18, 2009.

[12] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456. IEEE, 2015.

[13] Wikibooks. X86 assembly — wikibooks, the free textbook project, 2018. URL `https://en.wikibooks.org/w/index.php?title=X86_Assembly&oldid=3477563`. [Online; accessed 19-March-2019].

3

The C Programming Language

> If you want to teach systems, don't drum up the programmers, sort the issues, and make PRs. Instead, teach them to yearn for the vast and endless C.
>
> Antoine de Saint-Exupéry (With edits from Bhuvy)

Note: This chapter is long and goes into a lot of detail. Feel free to gloss over parts with which you have experience in.

C is the de-facto programming language to do serious system serious programming. Why? Most kernels have their accessible through C. The [7] and the XNU kernel Inc. [4] of which is based on are written in C and have C API - Application Programming Interface. The uses C++, but doing system programming on that is much harder on windows that for novice system programmers. C doesn't have like classes and (RAII) to clean up memory. C also gives you much more of an opportunity to shoot yourself in the foot, but it lets you do things at a much more fine-grained level.

History of C

C was developed by and at back in 1973 [8]. Back then, we had gems of programming languages like , , and . The goal of C was two-fold. Firstly, it was made to target the most popular computers at the time, such as the . Secondly, it tried to remove some of the lower-level constructs (managing , and programming assembly for), and create a language that had the power to express programs procedurally (as opposed to mathematically like LISP) with readable code. All this while still having the ability to interface with the operating system. It sounded like a tough feat. At first, it was only used internally at Bell Labs along with the UNIX operating system.

The first "real" standardization was with and Dennis Ritchie's book [6]. It is still widely regarded today as the only Portable set of C instructions. The K&R book is known as the de-facto standard for learning C. There were different standards of C from to , though ISO largely won out as a language specification. We will be mainly focusing on is the POSIX C library which extends ISO. Now to get the elephant out of the room, the is fails to be POSIX compliant. Mostly, this is so because the Linux developers didn't want to pay the fee for compliance. It is also because they did not want to be fully compliant with a multitude of different standards because that meant increased development costs to maintain compliance.

We will aim to use C99, as it is the standard that most computers recognize, but sometimes use some of the newer C11 features. We will also talk about some off-hand features like `getline` because they are so widely used with the . We'll begin by providing a fairly comprehensive overview of the language with language facilities. Feel free to gloss over if you have already worked with a C based language.

Features

- Speed. There is little separating a program and the system.
- Simplicity. C and its standard library comprise a simple set of portable functions.
- Manual Memory Management. C gives a program the ability to manage its memory. However, this can be a downside if a program has memory errors.
- Ubiquity. Through foreign function interfaces (FFI) and language bindings of various types, most other languages can call C functions and vice versa. The standard library is also everywhere. C has stood the test of time as a popular language, and it doesn't look like it is going anywhere.

Crash course introduction to C

The canonical way to start learning C is by starting with the program. The original example that Kernighan and Ritchie proposed way back when hasn't changed.

```
#include <stdio.h>
int main(void) {
  printf("Hello World\n");
  return 0;
}
```

1. The `#include` directive takes the file `stdio.h` (which stands for **st**an**d**ard **i**nput and **o**utput) located somewhere in your operating system, copies the text, and substitutes it where the `#include` was.
2. The `int main(void)` is a function declaration. The first word `int` tells the compiler the return type of the function. The part before the parenthesis (`main`) is the function name. In C, no two functions can have the same name in a single compiled program, although shared libraries may be able. Then, the parameter list comes after. When we provide the parameter list for regular functions `(void)` that means that the compiler should produce an error if the function is called with a non-zero number of arguments. For regular functions having a declaration like `void func()` means that the function can be called like `func(1, 2, 3)`, because there is no delimiter. `main` is a special function. There are many ways of declaring `main` but the standard ones are `int main(void)`, `int main()`, and `int main(int argc, char *argv[])`.
3. `printf("Hello World");` is what a function call. `printf` is defined as a part of `stdio.h`. The function has been compiled and lives somewhere else on our machine - the location of the C standard library. Just remember to include the header and call the function with the appropriate parameters (a string literal `"Hello World"`). If the newline isn't included, the buffer will not be flushed (i.e. the write will not complete immediately).

4. `return 0`. `main` has to return an integer. By convention, `return 0` means success and anything else means failure. Here are some exit codes / statuses with special meaning: `http://tldp.org/LDP/abs/html/exitcodes.html`. In general, assume 0 means success.

```
$ gcc main.c -o main
$ ./main
Hello World
$
```

1. `gcc` is short for the which has a host of compilers ready for use. The compiler infers from the extension that you are trying to compile a .c file.
2. `./main` tells your shell to execute the program in the current directory called main. The program then prints out "hello world".

If systems programming was as easy as writing hello world though, our jobs would be much easier.

Preprocessor

What is the ? Preprocessing is a copy and paste operation that the compiler performs **before** actually compiling the program. The following is an example of substitution

```
// Before preprocessing
#define MAX_LENGTH 10
char buffer[MAX_LENGTH]

// After preprocessing
char buffer[10]
```

There are side effects to the preprocessor though. One problem is that the preprocessor needs to be able to properly, meaning trying to redefine the internals of the C language with a preprocessor may be impossible. Another problem is that they can't be infinitely - there is a bounded depth where they need to stop. Macros are also simple text substitutions, without semantics. For example, look at what can happen if a macro tries to perform an inline modification.

```
#define min(a,b) a < b ? a : b
int main() {
  int x = 4;
  if(min(x++, 5)) printf("%d is six", x);
  return 0;
}
```

Macros are simple text substitution so the above example expands to

```
x++ < 100 ? x++ : 100
```

In this case, it is opaque what gets printed out, but it will be 6. Can you try to figure out why? Also, consider the edge case when comes into play.

```
int x = 99;
int r = 10 + min(99, 100); // r is 100!
// This is what it is expanded to
int r = 10 + 99 < 100 ? 99 : 100
// Which means
int r = (10 + 99) < 100 ? 99 : 100
```

There are also logical problems with the flexibility of certain parameters. One common source of confusion is with static arrays and the `sizeof` operator.

```
#define ARRAY_LENGTH(A) (sizeof((A)) / sizeof((A)[0]))
int static_array[10]; // ARRAY_LENGTH(static_array) = 10
int* dynamic_array = malloc(10); // ARRAY_LENGTH(dynamic_array) =
    2 or 1 consistently
```

What is wrong with the macro? Well, it works if a is passed in because `sizeof` a static array returns the number of bytes that array takes up and dividing it by the `sizeof(an_element)` would give the number of entries. But if passed a pointer to a piece of memory, taking the sizeof the pointer and dividing it by the size of the first entry won't always give us the size of the array.

Extra: Includes and conditionals

The other preprocessor include is the `#include` directive and conditionals. The include directive is explained by example.

```
// foo.h
int bar();
```

This is our file `bar.c` unpreprocessed.

```
#include "foo.h"
```

```c
int bar() {
}
```

After preprocessing, the compiler sees this

```c
// foo.c unpreprocessed
int bar();

int bar() {

}
```

The other tool is . If a macro is defined or truthy, that branch is taken.

```c
int main() {
  #ifdef __GNUC__
  return 1;
  #else
  return 0;
  #endif
}
```

Using gcc your compiler would preprocess the source to the following.

```c
int main() {
  return 1;
}
```

Using clang your compiler would preprocess to this.

```c
int main() {
  return 0;
}
```

Language Facilities

Keywords

C has an assortment of keywords. Here are some constructs that you should know briefly as of C99.

1. `break` is a keyword that is used in case statements or looping statements. When used in a case statement, the program jumps to the end of the block.

   ```
   switch(1) {
     case 1: /* Goes to this switch */
       puts("1");
       break; /* Jumps to the end of the block */
     case 2: /* Ignores this program */
       puts("2");
       break;
   } /* Continues here */
   ```

   In the context of a loop, using it breaks out of the inner-most loop. The loop can be either a `for`, `while`, or `do-while` construct

   ```
   while(1) {
     while(2) {
       break; /* Breaks out of while(2) */
     } /* Jumps here */
     break; /* Breaks out of while(1) */
   } /* Continues here */
   ```

2. `const` is a language level construct that tells the compiler that this data should remain constant. If one tries to change a const variable, the program will fail to compile. `const` works a little differently when put before the type, the compiler re-orders the first type and const. Then the compiler uses a left associativity rule. Meaning that whatever is left of the pointer is constant. This is known as const-correctness.

   ```
   const int i = 0; // Same as "int const i = 0"
   char *str = ...; // Mutable pointer to a mutable string
   const char *const_str = ...; // Mutable pointer to a constant
       string
   char const *const_str2 = ...; // Same as above
   const char *const const_ptr_str = ...;
   ```

```
// Constant pointer to a constant string
```

But, it is important to know that this is a compiler imposed restriction only. There are ways of getting around this, and the program will run fine with defined behavior. In systems programming, the only type of memory that you can't write to is system write-protected memory.

```
const int i = 0; // Same as "int const i = 0"
(*((int *)&i)) = 1; // i == 1 now
const char *ptr = "hi";
*ptr = ’\0’; // Will cause a Segmentation Violation
```

3. `continue` is a control flow statement that exists only in loop constructions. Continue will skip the rest of the loop body and set the program counter back to the start of the loop before.

```
int i = 10;
while(i--) {
  if(1) continue; /* This gets triggered */
  *((int *)NULL) = 0;
} /* Then reaches the end of the while loop */
```

4. `do {} while();` is another loop construct. These loops execute the body and then check the condition at the bottom of the loop. If the condition is zero, the next statement is executed – the program counter is set to the first instruction after the loop. Otherwise, the loop body is executed.

```
int i = 1;
do {
  printf("%d\n", i--);
} while (i > 10) /* Only executed once */
```

5. `enum` is to declare an enumeration. An enumeration is a type that can take on many, finite values. If you have an enum and don't specify any numerics, the C compiler will generate a unique number for that enum (within the context of the current enum) and use that for comparisons. The syntax to declare an instance of an enum is `enum <type> varname`. The added benefit to this is that the compiler can type check these expressions to make sure that you are only comparing alike types.

```
enum day{ monday, tuesday, wednesday,
  thursday, friday, saturday, sunday};
```

```c
void process_day(enum day foo) {
  switch(foo) {
    case monday:
      printf("Go home!\n"); break;
    // ...
  }
}
```

It is completely possible to assign enum values to either be different or the same. It is not advisable to rely on the compiler for consistent numbering, if you assign numbers. If you are going to use this abstraction, try not to break it.

```c
enum day{
  monday = 0,
  tuesday = 0,
  wednesday = 0,
  thursday = 1,
  friday = 10,
  saturday = 10,
  sunday = 0};

void process_day(enum day foo) {
  switch(foo) {
    case monday:
      printf("Go home!\n"); break;
    // ...
  }
}
```

6. `extern` is a special keyword that tells the compiler that the variable may be defined in another object file or a library, so the program compiles on missing variable because the program will reference a variable in the system or another file.

```c
// file1.c
extern int panic;

void foo() {
  if (panic) {
    printf("NONONONONO");
  } else {
    printf("This is fine");
  }
}
```

```
//file2.c

int panic = 1;
```

7. for is a keyword that allows you to iterate with an initialization condition, a loop invariant, and an update condition. This is meant to be equivalent to a while loop, but with differing syntax.

   ```
   for (initialization; check; update) {
     //...
   }

   // Typically
   int i;
   for (i = 0; i < 10; i++) {
     //...
   }
   ```

   As of the C89 standard, one cannot declare variables inside the for loop initialization block. This is because there was a disagreement in the standard for how the scoping rules of a variable defined in the loop would work. It has since been resolved with more recent standards, so people can use the for loop that they know and love today

   ```
   for(int i = 0; i < 10; ++i) {
   ```

   The order of evaluation for a for loop is as follows

   (a) Perform the initialization statement.

   (b) Check the invariant. If false, terminate the loop and execute the next statement. If true, continue to the body of the loop.

   (c) Perform the body of the loop.

   (d) Perform the update statement.

   (e) Jump to checking the invariant step.

8. goto is a keyword that allows you to do conditional jumps. Do not use goto in your programs. The reason being is that it makes your code infinitely more hard to understand when strung together with multiple chains, which is called spaghetti code. It is acceptable to use in some contexts though, for example, error checking code in the Linux kernel. The keyword is usually used in kernel contexts when adding another stack frame for cleanup isn't a good idea. The canonical example of kernel cleanup is as below.

   ```
   void setup(void) {
   ```

```
Doe *deer;
Ray *drop;
Mi *myself;

if (!setupdoe(deer)) {
  goto finish;
}

if (!setupray(drop)) {
  goto cleanupdoe;
}

if (!setupmi(myself)) {
  goto cleanupray;
}

perform_action(deer, drop, myself);

cleanupray:
cleanup(drop);
cleanupdoe:
cleanup(deer);
finish:
return;
}
```

9. `if else else-if` are control flow keywords. There are a few ways to use these (1) A bare if (2) An if with an else (3) an if with an else-if (4) an if with an else if and else. Note that an else is matched with the most recent if. A subtle bug related to a mismatched if and else statement, is the dangling else problem. The statements are always executed from the if to the else. If any of the intermediate conditions are true, the if block performs that action and goes to the end of that block.

```
// (1)

if (connect(...))
  return -1;

// (2)
if (connect(...)) {
  exit(-1);
} else {
  printf("Connected!");
}

// (3)
if (connect(...)) {
  exit(-1);
} else if (bind(..)) {
  exit(-2);
}

// (1)
if (connect(...)) {
  exit(-1);
} else if (bind(..)) {
  exit(-2);
} else {
  printf("Successfully bound!");
}
```

10. inline is a compiler keyword that tells the compiler it's okay to moit the C function call procedure and "paste" the code in the callee. Instead, the compiler is hinted at substituting the function body directly into the calling function. This is not always recommended explicitly as the compiler is usually smart enough to know when to inline a function for you.

```
inline int max(int a, int b) {
  return a < b ? a : b;
}

int main() {
  printf("Max %d", max(a, b));
  // printf("Max %d", a < b ? a : b);
}
```

11. restrict is a keyword that tells the compiler that this particular memory region shouldn't overlap with all other memory regions. The use case for this is to tell users of the program that it is undefined behavior if the memory regions overlap. Note that memcpy has undefined behavior when memory regions overlap. If this might be the case in your program, consider using memmove.

```
memcpy(void * restrict dest, const void* restrict src, size_t
    bytes);

void add_array(int *a, int * restrict c) {
  *a += *c;
}
int *a = malloc(3*sizeof(*a));
*a = 1; *a = 2; *a = 3;
add_array(a + 1, a) // Well defined
add_array(a, a) // Undefined
```

12. return is a control flow operator that exits the current function. If the function is void then it simply exits the functions. Otherwise, another parameter follows as the return value.

```c
void process() {
  if (connect(...)) {
    return -1;
  } else if (bind(...)) {
    return -2
  }
  return 0;
}
```

13. signed is a modifier which is rarely used, but it forces a type to be signed instead of unsigned. The reason that this is so rarely used is because types are signed by default and need to have the unsigned modifier to make them unsigned but it may be useful in cases where you want the compiler to default to a signed type such as below.

```c
int count_bits_and_sign(signed representation) {
  //...
}
```

14. sizeof is an operator that is evaluated at compile-time, which evaluates to the number of bytes that the expression contains. When the compiler infers the type the following code changes as follows.

```c
char a = 0;
printf("%zu", sizeof(a++));
```

```c
char a = 0;
printf("%zu", 1);
```

Which then the compiler is allowed to operate on further. The compiler must have a complete definition of the type at compile-time - not link time - or else you may get an odd error. Consider the following

```c
// file.c
```

```
struct person;

printf("%zu", sizeof(person));

// file2.c

struct person {
  // Declarations
}
```

This code will not compile because sizeof is not able to compile `file.c` without knowing the full declaration of the `person` struct. That is typically why programmers either put the full declaration in a header file or we abstract the creation and the interaction away so that users cannot access the internals of our struct. Additionally, if the compiler knows the full length of an array object, it will use that in the expression instead of having it decay into a pointer.

```
char str1[] = "will be 11";
char* str2 = "will be 8";
sizeof(str1) //11 because it is an array
sizeof(str2) //8 because it is a pointer
```

Be careful, using sizeof for the length of a string!

15. `static` is a type specifier with three meanings.

    (a) When used with a global variable or function declaration it means that the scope of the variable or the function is only limited to the file.

    (b) When used with a function variable, that declares that the variable has static allocation – meaning that the variable is allocated once at program startup not every time the program is run, and its lifetime is extended to that of the program.

```
// visible to this file only
static int i = 0;

static int _perform_calculation(void) {
  // ...
}

char *print_time(void) {
  static char buffer[200]; // Shared every time a function is
      called
  // ...
}
```

16. struct is a keyword that allows you to pair multiple types together into a new structure. C-structs are contiguous regions of memory that one can access specific elements of each memory as if they were separate variables. Note that there might be padding between elements, such that each variable is memory-aligned (starts at a memory address that is a multiple of its size).

    ```
    struct hostname {
      const char *port;
      const char *name;
      const char *resource;
    }; // You need the semicolon at the end
    // Assign each individually
    struct hostname facebook;
    facebook.port = "80";
    facebook.name = "www.google.com";
    facebook.resource = "/";

    // You can use static initialization in later versions of c
    struct hostname google = {"80", "www.google.com", "/"};
    ```

17. switch case default Switches are essentially glorified jump statements. Meaning that you take either a byte or an integer and the control flow of the program jumps to that location. Note that, the various cases of a switch statement fall through. It means that if execution starts in one case, the flow of control will continue to all subsequent cases, until a break statement.

    ```
    switch(/* char or int */) {
      case INT1: puts("1");
      case INT2: puts("2");
      case INT3: puts("3");
    }
    ```

    If we give a value of 2 then

    ```
    switch(2) {
      case 1: puts("1"); /* Doesn’t run this */
      case 2: puts("2"); /* Runs this */
      case 3: puts("3"); /* Also runs this */
    }
    ```

One of the more famous examples of this is Duff's device which allows for loop unrolling. You don't need to understand this code for the purposes of this class, but it is fun to look at [2].

```
send(to, from, count)
register short *to, *from;
register count;
{
  register n=(count+7)/8;
  switch(count%8){
  case 0:  do{ *to = *from++;
  case 7:      *to = *from++;
  case 6:      *to = *from++;
  case 5:      *to = *from++;
  case 4:      *to = *from++;
  case 3:      *to = *from++;
  case 2:      *to = *from++;
  case 1:      *to = *from++;
    }while(--n>0);
  }
}
```

This piece of code highlights that switch statements are goto statements, and you can put any code on the other end of a switch case. Most of the time it doesn't make sense, some of the time it just makes too much sense.

18. typedef declares an alias for a type. Often used with structs to reduce the visual clutter of having to write 'struct' as part of the type.

```
typedef float real;
real gravity = 10;
// Also typedef gives us an abstraction over the underlying
    type used.
// In the future, we only need to change this typedef if we
// wanted our physics library to use doubles instead of floats.

typedef struct link link_t;
//With structs, include the keyword 'struct' as part of the
    original types
```

In this class, we regularly typedef functions. A typedef for a function can be this for example

```
typedef int (*comparator)(void*,void*);
```

```c
int greater_than(void* a, void* b){
    return a > b;
}
comparator gt = greater_than;
```

This declares a function type comparator that accepts two `void*` params and returns an integer.

19. `union` is a new type specifier. A union is one piece of memory that many variables occupy. It is used to maintain consistency while having the flexibility to switch between types without maintaining functions to keep track of the bits. Consider an example where we have different pixel values.

```c
union pixel {
  struct values {
    char red;
    char blue;
    char green;
    char alpha;
  } values;
  uint32_t encoded;
}; // Ending semicolon needed
union pixel a;
// When modifying or reading
a.values.red;
a.values.blue = 0x0;

// When writing to a file
fprintf(picture, "%d", a.encoded);
```

20. `unsigned` is a type modifier that forces `unsigned` behavior in the variables they modify. Unsigned can only be used with primitive int types (like `int` and `long`). There is a lot of behavior associated with unsigned arithmetic. For the most part, unless your code involves bit shifting, it isn't essential to know the difference in behavior with regards to unsigned and signed arithmetic.

21. `void` is a double meaning keyword. When used in terms of function or parameter definition, it means that the function explicitly returns no value or accepts no parameter, respectively. The following declares a function that accepts no parameters and returns nothing.

```c
void foo(void);
```

The other use of `void` is when you are defining an `lvalue`. A `void *` pointer is just a memory address. It is specified as an incomplete type meaning that you cannot dereference it but it can be promoted to any time to any other type. Pointer arithmetic with this pointer is undefined behavior.

```c
int *array = void_ptr; // No cast needed
```

22. volatile is a compiler keyword. This means that the compiler should not optimize its value out. Consider the following simple function.

    ```c
    int flag = 1;
    pass_flag(&flag);
    while(flag) {
        // Do things unrelated to flag
    }
    ```

    The compiler may, since the internals of the while loop have nothing to do with the flag, optimize it to the following even though a function may alter the data.

    ```c
    while(1) {
        // Do things unrelated to flag
    }
    ```

    If you use the volatile keyword, the compiler is forced to keep the variable in and perform that check. This is useful for cases where you are doing multi-process or multi-threaded programs so that we can affect the running of one sequence of execution with another.

23. while represents the traditional while loop. There is a condition at the top of the loop, which is checked before every execution of the loop body. If the condition evaluates to a non-zero value, the loop body will be run.

C data types

There are many data types in C. As you may realize, all of them are either integers or floating point numbers and other types are variations of these.

1. char Represents exactly one byte of data. The number of bits in a byte might vary. unsigned char and signed char means the exact same thing. This must be aligned on a boundary (meaning you cannot use bits in between two addresses). The rest of the types will assume 8 bits in a byte.

2. short (short int) must be at least two bytes. This is aligned on a two byte boundary, meaning that the address must be divisible by two.

3. int must be at least two bytes. Again aligned to a two byte boundary [5, P. 34]. On most machines this will be 4 bytes.

4. `long (long int)` must be at least four bytes, which are aligned to a four byte boundary. On some machines this can be 8 bytes.

5. `long long` must be at least eight bytes, aligned to an eight byte boundary.

6. `float` represents an IEEE-754 single precision floating point number tightly specified by IEEE [1]. This will be four bytes aligned to a four byte boundary on most machines.

7. `double` represents an IEEE-754 double precision floating point number specified by the same standard, which is aligned to the nearest eight byte boundary.

If you want a fixed width integer type, for more portable code, you may use the types defined in stdint.h, which are of the form [u]int*width*_t, where u (which is optional) represents the signedness, and width is any of 8, 16, 32, and 64.

Operators

Operators are language constructs in C that are defined as part of the grammar of the language. These operators are listed in order of precedence.

- `[]` is the subscript operator. `a[n] == (a + n)*` where `n` is a number type and `a` is a pointer type.

- `->` is the structure dereference (or arrow) operator. If you have a pointer to a struct `*p`, you can use this to access one of its elements. `p->element`.

- `.` is the structure reference operator. If you have an object `a` then you can access an element `a.element`.

- `+/-a` is the unary plus and minus operator. They either keep or negate the sign, respectively, of the integer or float type underneath.

- `*a` is the dereference operator. If you have a pointer `*p`, you can use this to access the element located at this memory address. If you are reading, the return value will be the size of the underlying type. If you are writing, the value will be written with an offset.

- `&a` is the address-of operator. This takes an element and returns its address.

- `++` is the increment operator. You can use it as a prefix or postfix, meaning that the variable that is being incremented can either be before or after the operator. `a = 0; ++a === 1` and `a = 1; a++ === 0`.

- `-` is the decrement operator. This has the same semantics as the increment operator except that it decreases the value of the variable by one.

- `sizeof` is the sizeof operator, that is evaluated at the time of compilation. This is also mentioned in the keywords section.

- `a <mop> b` where `<mop> in {+, -, *, %, /}` are the arithmetic binary operators. If the operands are both number types, then the operations are plus, minus, times, modulo, and division respectively. If the left operand is a pointer and the right operand is an integer type, then only plus or minus may be used and the rules for pointer arithmetic are invoked.

- `»/«` are the bit shift operators. The operand on the right has to be an integer type whose signedness is ignored unless it is signed negative in which case the behavior is undefined. The operator on the left decides a lot of semantics. If we are left shifting, there will always be zeros introduced on the right. If we are right shifting there are a few different cases

  - If the operand on the left is signed, then the integer is sign-extended. This means that if the number has the sign bit set, then any shift right will introduce ones on the left. If the number does not have the sign bit set, any shift right will introduce zeros on the left.
  - If the operand is unsigned, zeros will be introduced on the left either way.

```
unsigned short uns = -127; // 1111111110000001
short sig = 1; // 0000000000000001
uns << 2; // 1111111000000100
sig << 2; // 0000000000000100
uns >> 2; // 1111111111100000
sig >> 2; // 0000000000000000
```

Note that shifting by the word size (e.g. by 64 in a 64-bit architecture) results in undefined behavior.

- `<=/>=` are the greater than equal to/less than equal to, relational operators. They work as their name implies.
- `</>` are the greater than/less than relational operators. They again do as the name implies.
- `==/=` are the equal/not equal to relational operators. They once again do as the name implies.
- `&&` is the logical AND operator. If the first operand is zero, the second won't be evaluated and the expression will evaluate to 0. Otherwise, it yields a 1-0 value of the second operand.
- `||` is the logical OR operator. If the first operand is not zero, then second won't be evaluated and the expression will evaluate to 1. Otherwise, it yields a 1-0 value of the second operand.
- `!` is the logical NOT operator. If the operand is zero, then this will return 1. Otherwise, it will return 0.
- `&` is the bitwise AND operator. If a bit is set in both operands, it is set in the output. Otherwise, it is not.
- `|` is the bitwise OR operator. If a bit is set in either operand, it is set in the output. Otherwise, it is not.
- is the bitwise NOT operator. If a bit is set in the input, it will not be set in the output and vice versa.
- `?:` is the ternary / conditional operator. You put a boolean condition before the and if it evaluates to non-zero the element before the colon is returned otherwise the element after is. `1 ?  a :  b === a` and `0 ?  a :  b === b`.
- `a, b` is the comma operator. `a` is evaluated and then `b` is evaluated and `b` is returned. In a sequence of multiple statements delimited by commas, all statements are evaluated from left to right, and the right-most expression is returned.

The C and Linux

Up until this point, we've covered C's language fundamentals. We'll now be focusing our attention to C and the POSIX variety of functions available to us to interact with the operating systems. We will talk about portable functions, for example fwrite printf. We will be evaluating the internals and scrutinizing them under the POSIX models and more specifically GNU/Linux. There are several things to that philosophy that makes the rest of this easier to know, so we'll put those things here.

Everything is a file

One POSIX mantra is that everything is a file. Although that has become recently outdated, and moreover wrong, it is the convention we still use today. What this statement means is that everything is a file descriptor, which is an integer. For example, here is a file object, a network socket, and a kernel object. These are all references to records in the kernel's file descriptor table.

```
int file_fd = open(...);
int network_fd = socket(...);
int kernel_fd = epoll_create1(...);
```

And operations on those objects are done through system calls. One last thing to note before we move on is that the file descriptors are merely *pointers*. Imagine that each of the file descriptors in the example actually refers to an entry in a table of objects that the operating system picks and chooses from (that is, the file descriptor table). Objects can be allocated and deallocated, closed and opened, etc. The program interacts with these objects by using the API specified through system calls, and library functions.

System Calls

Before we dive into common C functions, we need to know what a system call is. If you are a student and have completed HW0, feel free to gloss over this section.

A system call is an operation that the kernel carries out. First, the operating system prepares a system call. Next, the kernel executes the system call to the best of its ability in kernel space and is a privileged operation. In the previous example, we got access to a file descriptor object. We can now also write some bytes to the file descriptor object that represents a file, and the operating system will do its best to get the bytes written to the disk.

```
write(file_fd, "Hello!", 6);
```

When we say the kernel tries its best, this includes the possibility that the operation could fail for several reasons. Some of them are: the file is no longer valid, the hard drive failed, the system was interrupted etc. The way that a programmer communicates with the outside system is with system calls. An important thing to note is that system calls are expensive. Their cost in terms of time and CPU cycles has recently been decreased, but try to use them as sparingly as possible.

C System Calls

Many C functions that will be discussed in the next sections are abstractions that call the correct underlying system call, based on the current platform. Their Windows implementation, for example, may be entirely different from that of other operating systems. Nevertheless, we will be studying these in the context of their Linux implementation.

Common C Functions

To find more information about any functions, please use the man pages. Note the man pages are organized into sections. Section 2 are System calls. Section 3 are C libraries. On the web, Google `man 7 open`. In the shell, `man -S2 open` or `man -S3 printf`

Handling Errors

Before we get into the nitty gritty of all the functions, know that most functions in C handle errors return oriented. This is at odds with programming languages like C++ or Java where the errors are handled with exceptions. There are a number of arguments against exceptions.

1. Exceptions make control flow harder to understand.
2. Exception oriented languages need to keep stack traces and maintain jump tables.
3. Exceptions may be complex objects.

There are a few arguments for exceptions as well

1. Exceptions can come from several layers deep.
2. Exceptions help reduce global state.
3. Exceptions differentiate business logic and normal flow.

Whatever the pros/cons are, we use the former because of backwards compatibility with languages like FORTRAN [3, P. 84]. Each thread will get a copy of `errno` because it is stored at the top of each thread's stack – more on threads later. One makes a call to a function that could return an error and if that function returns an error according to the man pages, it is up to the programmer to check errno.

```
#include <errno.h>

FILE *f = fopen("/does/not/exist.txt", "r");
if (NULL == f) {
    fprintf(stderr, "Errno is %d\n", errno);
    fprintf(stderr, "Description is %s\n", strerror(errno));
}
```

There is a shortcut function perror that prints the english description of errno. Also, a function may return the error code in the return value itself.

```
int s = getnameinfo(...);
if (0 != s) {
     fprintf(stderr, "getnameinfo: %s\n", gai_strerror(s));
}
```

Be sure to check the man page for return code characteristics.

Input / Output

In this section we will cover all the basic input and output functions in the standard library with references to system calls. Every process has three streams of data when it starts execution: standard input (for program input), standard output (for program output), and standard error (for error and debug messages). Usually, standard input is sourced from the terminal in which the program is being run in, and standard out is the same terminal. However, a programmer can use redirection such that their program can send output and/or receive input, to and from a file, or other programs.

They are designated by the file descriptors 0 and 1 respectively. 2 is reserved for standard error which by library convention is unbuffered (i.e. IO operations are performed immediately).

stdout oriented streams

Standard output or stdout oriented streams are streams whose only options are to write to stdout. printf is the function with which most people are familiar in this category. The first parameter is a format string that includes placeholders for the data to be printed. Common format specifiers are the following

1. %s treat the argument as a c string pointer, keep printing all characters until the NULL-character is reached

2. %d prints the argument as an integer

3. %p print the argument as a memory address.

For performance, printf buffers data until its cache is full or a newline is printed. Here is an example of printing things out.

```
char *name = ... ; int score = ...;
printf("Hello %s, your result is %d\n", name, score);
printf("Debug: The string and int are stored at: %p and %p\n",
    name, &score );
// name already is a char pointer and points to the start of the
    array.
// We need "&" to get the address of the int variable
```

From the previous section, `printf` calls the system call `write`. `printf` is a C library function, while `write` is a system call system.

The buffering semantics of printf is a little complicated. ISO defines three types of streams [5, P. 278]

- Unbuffered, where the contents of the stream reach their destination as soon as possible.
- Line Buffered, where the contents of the stream reach their destination as soon as a newline is provided.
- Fully Buffered, where the contents of the stream reach their destination as soon as the buffer is full.

Standard Error is defined as "not fully buffered" [5, P. 279]. Standard Output and Input are merely defined to be fully buffered if and only if the stream destination is not an interactive device. Usually, standard error will be unbuffered, standard input and output will be line buffered if the output is a terminal otherwise fully buffered. This relates to printf because printf merely uses the abstraction provided by the FILE interface and uses the above semantics to determine when to write. One can force a write by calling fflush() on the stream.

To print strings and single characters, use `puts(char *name )` and `putchar(char c )`

```
puts("Current selection: ");
putchar('1');
```

Other streams

To print to other file streams, use `fprintf( _file_ , "Hello %s, score: %d", name, score);` Where _file_ is either predefined ('stdout' or 'stderr') or a FILE pointer that was returned by `fopen` or `fdopen`. There is a printf equivalent that works with file descriptors, called dprintf. Just use `dprintf(int fd, char* format_string, ...);`.

To print data into a C string, use `sprintf` or better `snprintf`. `snprintf` returns the number of characters written excluding the terminating byte. We would use `sprintf` the size of the printed string is less than the provided buffer – think about printing an integer, it will never be more than 11 characters with the NUL byte. If printf is dealing with variadic input, it is safer to use the former function as shown in the following snippet.

```
// Fixed
char int_string[20];
sprintf(int_string, "%d", integer);

// Variable length
char result[200];
int len = snprintf(result, sizeof(result), "%s:%d", name, score);
```

stdin oriented functions

Standard input or stdin oriented functions read from stdin directly. Most of these functions have been deprecated due to them being poorly designed. These functions treat stdin as a file from which we can read bytes. One of the most notorious offenders is gets. gets is deprecated in C99 standard and has been removed from the latest C standard (C11). The reason that it was deprecated was that there is no way to control the length being read, therefore buffers could get overrun easily. When this is done maliciously to hijack program control flow, this is known as a buffer overflow.

Programs should use fgets or getline instead. Here is a quick example of reading at most 10 characters from standard input.

```
char *fgets (char *str, int num, FILE *stream);

ssize_t getline(char **lineptr, size_t *n, FILE *stream);

// Example, the following will not read more than 9 chars
char buffer[10];
char *result = fgets(buffer, sizeof(buffer), stdin);
```

Note that, unlike gets, fgets copies the newline into the buffer. On the other hand, one of the advantages of getline is that will automatically allocate and reallocate a buffer on the heap of sufficient size.

```
// ssize_t getline(char **lineptr, size_t *n, FILE *stream);

/* set buffer and size to 0; they will be changed by getline */
char *buffer = NULL;
size_t size = 0;

ssize_t chars = getline(&buffer, &size, stdin);

// Discard newline character if it is present,
if (chars > 0 && buffer[chars-1] == ’\n’)
buffer[chars-1] = ’\0’;

// Read another line.
// The existing buffer will be re-used, or, if necessary,
// It will be ‘free‘’d and a new larger buffer will ‘malloc‘’d
chars = getline(&buffer, &size, stdin);

// Later... don’t forget to free the buffer!
free(buffer);
```

In addition to those functions, we have perror that has a two-fold meaning. Let’s say that a function call failed using the errno convention. perror(const char* message) will print the English version of the error to stderr.

```
int main(){
  int ret = open("IDoNotExist.txt", O_RDONLY);
  if(ret < 0){
    perror("Opening IDoNotExist:");
  }
  //...
  return 0;
}
```

To have a library function parse input in addition to reading it, use `scanf` (or `fscanf` or `sscanf`) to get input from the default input stream, an arbitrary file stream or a C string, respectively. All of those functions will return how many items were parsed. It is a good idea to check if the number is equal to the amount expected. Also naturally like `printf`, `scanf` functions require valid pointers. Instead of pointing to valid memory, they need to also be writable. It's a common source of error to pass in an incorrect pointer value. For example,

```
int *data = malloc(sizeof(int));
char *line = "v 10";
char type;
// Good practice: Check scanf parsed the line and read two values:
int ok = 2 == sscanf(line, "%c %d", &type, &data); // pointer error
```

We wanted to write the character value into c and the integer value into the malloc'd memory. However, we passed the address of the data pointer, not what the pointer is pointing to! So `sscanf` will change the pointer itself. The pointer will now point to address 10 so this code will later fail when free(data) is called.

Now, scanf will keep reading characters until the string ends. To stop scanf from causing a buffer overflow, use a format specifier. Make sure to pass one less than the size of the buffer.

```
char buffer[10];
scanf("%9s", buffer); // reads up to 9 characters from input
    (leave room for the 10th byte to be the terminating byte)
```

One last thing to note is if system calls are expensive, the `scanf` family is much more expensive due to compatibility reasons. Since it needs to be able to process all of the printf specifiers correctly, the code isn't efficient TODO: citation needed. For highly performant programs, one should write the parsing themselves. If it is a one-off program or script, feel free to use scanf.

string.h

String.h functions are a series of functions that deal with how to manipulate and check pieces of memory. Most of them deal with C-strings. A C-string is a series of bytes delimited by a NUL character which is equal to the byte 0x00. More information about all of these functions. Any behavior missing from the documentation, such as the result of `strlen(NULL)` is considered undefined behavior.

- `int strlen(const char *s)` returns the length of the string.

- `int strcmp(const char *s1, const char *s2)` returns an integer determining the lexicographic order of the strings. If s1 where to come before s2 in a dictionary, then a -1 is returned. If the two strings are equal, then 0. Else, 1.

- `char *strcpy(char *dest, const char *src)` Copies the string at `src` to `dest`. **This function assumes dest has enough space for src otherwise undefined behavior**

- `char *strcat(char *dest, const char *src)` Concatenates the string at `src` to the end of destination. **This function assumes that there is enough space for `src` at the end of destination including the NUL byte**

- `char *strdup(const char *dest)` Returns a `malloc`'d copy of the string.

- `char *strchr(const char *haystack, int needle)` Returns a pointer to the first occurrence of `needle` in the `haystack`. If none found, `NULL` is returned.

- `char *strstr(const char *haystack, const char *needle)` Same as above but this time a string!

- `char *strtok(const char *str, const char *delims)`

  A dangerous but useful function strtok takes a string and tokenizes it. Meaning that it will transform the strings into separate strings. This function has a lot of specs so please read the man pages a contrived example is below.

```c
#include <stdio.h>
#include <string.h>

int main(){
  char* upped = strdup("strtok,is,tricky,!!");
  char* start = strtok(upped, ",");
  do{
    printf("%s\n", start);
  }while((start = strtok(NULL, ",")));
  return 0;
}
```

Output

```
strtok
is
tricky
!!
```

Why is it tricky? Well what happens when upped is changed to the following?

```
char* upped = strdup("strtok,is,tricky,,,!!");
```

- For integer parsing use `long int strtol(const char *nptr, char **endptr, int base);` or `long long int strtoll(const char *nptr, char **endptr, int base);`.

  What these functions do is take the pointer to your string `*nptr` and a `base` (i.e. binary, octal, decimal, hexadecimal etc) and an optional pointer `endptr` and returns a parsed value.

  ```
  int main(){
    const char *nptr = "1A2436";
    char* endptr;
    long int result = strtol(nptr, &endptr, 16);
    return 0;
  }
  ```

  Be careful though! Error handling is tricky because the function won't return an error code. If passed an invalid number string, it will return 0. The caller has to be careful from a valid 0 and an error. This often involves an errno trampoline as shown below.

  ```
  int main(){
    const char *input = "0"; // or "!##@" or ""
    char* endptr;
    int saved_errno = errno;
    errno = 0
    long int parsed = strtol(input, &endptr, 10);
    if(parsed == 0 && errno != 0){
      // Definitely an error
    }
    errno = saved_errno;
    return 0;
  }
  ```

- `void *memcpy(void *dest, const void *src, size_t n)` moves `n` bytes starting at `src` to `dest`. **Be careful**, there is undefined behavior when the memory regions overlap. This is one of the classic "This works on my machine!" examples because many times Valgrind won't be able to pick it up because it will look like it works on your machine. Consider the safer version `memmove`.

- `void *memmove(void *dest, const void *src, size_t n)` does the same thing as above, but if the memory regions overlap then it is guaranteed that all the bytes will get copied over correctly. `memcpy` and `memmove` both in `string.h`?

C Memory Model

The C memory model is probably unlike most that you've seen before. Instead of allocating an object with type safety, we either use an automatic variable or request a sequence of bytes with malloc or another family member and later we free it.

Structs

In low-level terms, a struct is a piece of contiguous memory, nothing more. Just like an array, a struct has enough space to keep all of its members. But unlike an array, it can store different types. Consider the contact struct declared above.

```
struct contact {
  char firstname[20];
  char lastname[20];
  unsigned int phone;
};

struct contact bhuvan;
```

We will often use the following typedef, so we can write use the struct name as the full type.

```
typedef struct contact contact;
contact bhuvan;

typedef struct optional_name {
  ...
} contact;
```

If you compile the code without any optimizations and reordering, you can expect the addresses of each of the variables to look like this.

```
&bhuvan          // 0x100
&bhuvan.firstname // 0x100 = 0x100+0x00
&bhuvan.lastname // 0x114 = 0x100+0x14
&bhuvan.phone    // 0x128 = 0x100+0x28
```

All your compiler does is say "reserve this much space". Whenever a read or write occurs in the code, the compiler will calculate the offsets of the variable. The offsets are where the variable starts at. The phone variables starts at the 0x128th bytes and continues for sizeof(int) bytes with this compiler. **Offsets don't determine where the variable ends though**. Consider the following hack seen in a lot of kernel code.

```
typedef struct {
  int length;
  char c_str[0];
} string;

const char* to_convert = "bhuvan";
int length = strlen(to_convert);

// Let's convert to a c string
string* bhuvan_name;
bhuvan_name = malloc(sizeof(string) + length+1);
```

Currently, our memory looks like the following image. There is nothing in those boxes

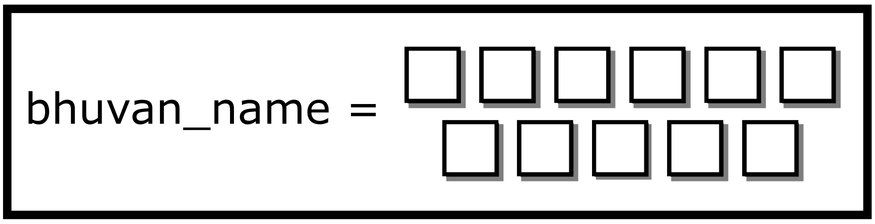


Figure 3.1: Struct pointing to 11 empty boxes

So what happens when we assign length? The first four boxes are filled with the value of the variable at length. The rest of the space is left untouched. We will assume that our machine is big endian. This means that the least significant byte is last.

```
bhuvan_name->length = length;
```

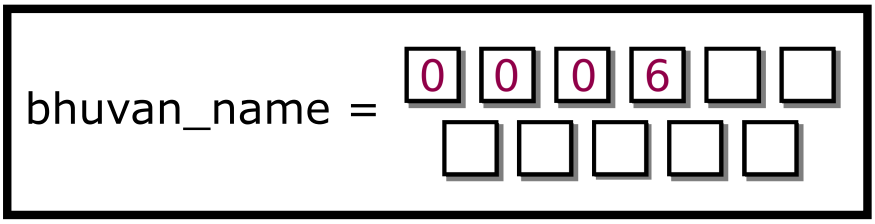


Figure 3.2: Struct pointing to 11 boxes, 4 filled with 0006, 7 junk

Now, we can write a string to the end of our struct with the following call.

m

```
strcpy(bhuvan_name->c_str, to_convert);
```

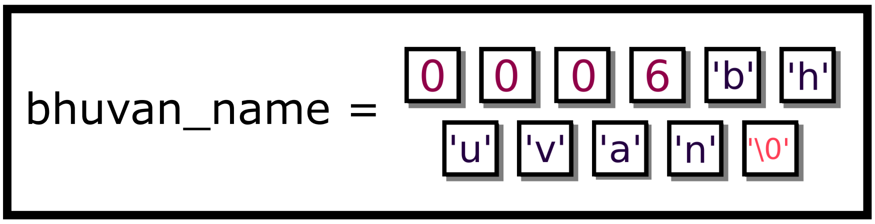


Figure 3.3: Struct pointing to 11 boxes, 4 filled with 0006, 7 the stirng “bhuvan”

We can even do a sanity check to make sure that the strings are equal.
m

```
strcmp(bhuvan_name->c_str, "bhuvan") == 0 //The strings are equal!
```

What that zero length array does is point to the **end of the struct** this means that the compiler will leave room for all of the elements calculated with respect to their size on the operating system (ints, chars, etc). The zero length array will take up no bytes of space. Since structs are continuous pieces of memory, we can allocate **more** space than required and use the extra space as a place to store extra bytes. Although this seems like a parlor trick, it is an important optimization because to have a variable length string any other way, one would need to have two different memory allocation calls. This is highly inefficient for doing something as common in programming as is string manipulation.

Extra: Struct packing

Structs may require something called padding (tutorial). **We do not expect you to pack structs in this course, know that compilers perform it**. This is because in the early days (and even now) loading an address in memory happens in 32-bit or 64-bit blocks. This also meant requested addresses had to be multiples of block sizes.

```
struct picture{
  int height;
  pixel** data;
  int width;
  char* encoding;
}
```

You think the picture looks like this. One box is four bytes.

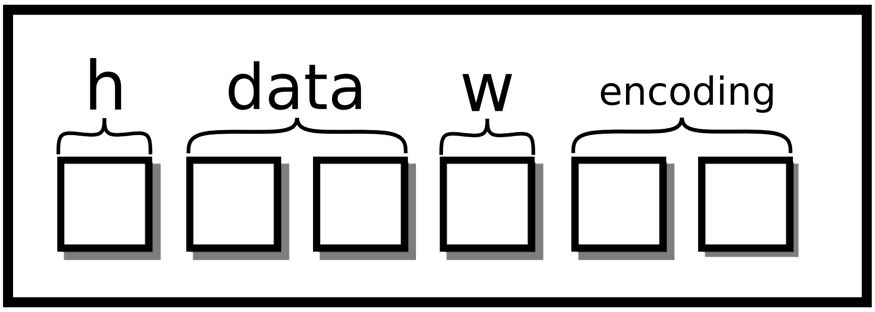


Figure 3.4: Six box struct

However, with struct packing, it would conceptually look like this:

```
struct picture{
  int height;
  char slop1[4];
  pixel** data;
  int width;
  char slop2[4];
  char* encoding;
}
```

Visually, we'd add two extra boxes to our diagram

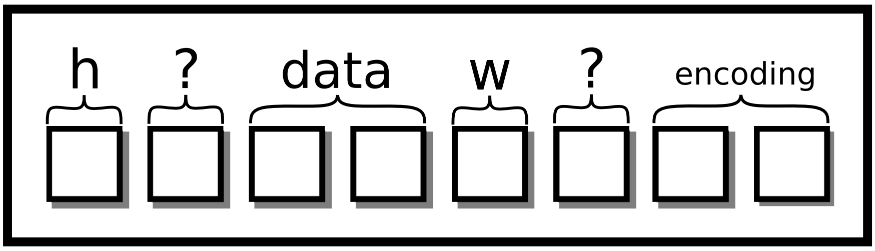


Figure 3.5: Eight box struct, two boxes of slop

This padding is common on a 64-bit system. Other time, a processor supports unaligned access, leaving the compiler able to pack structs. What does this mean? We can have a variable start at a non-64-bit boundary. The processor will figure out the rest. To enable this, set an attribute.

```
struct __attribute__((packed, aligned(4))) picture{
  int height;
  pixel** data;
  int width;
  char* encoding;
}
```

Now our figure will look like the clean struct as in figure 3.4 But now, every time the processor needs to access `data` or `encoding`, two memory accesses are required. A possible alternative is to reorder the struct.

```
struct picture{
  int height;
  int width;
  pixel** data;
  char* encoding;
}
```

Strings in C

In C, we have Null Terminated strings rather than Length Prefixed for historical reasons. For everyday programmers, remember to NUL terminate your string! A string in C is defined as a bunch of bytes ended by ‘’ or the NUL Byte.

Places for strings

Whenever you define a string literal - one in the form `char* str = "constant"` – that string is stored in the *data* section. Depending on your architecture, it is **read-only**, meaning that any attempt to modify the string will cause a SEGFAULT. One can also declare strings to be either in the writable data segment or the stack. To do so, specify a length for the string or put brackets instead of a pointer `char str[] = "mutable"` and put in the global scope or the function scope for the data segment or the stack respectively. If one, however, `malloc`'s space, one can change that string to be whatever they want. Forgetting to NUL terminate a string has a big effect on the strings! Bounds checking is important. The heartbleed bug mentioned earlier in the book is partially because of this.

Strings in C are represented as characters in memory. The end of the string includes a NUL (0) byte. So "ABC" requires four(4) bytes. The only way to find out the length of a C string is to keep reading memory until you find the NUL byte. C characters are always exactly one byte each.

String literals are constant

A string literal is naturally constant. Any write will cause the operating system to produce a SEGFAULT.

```
char array[] = "Hi!"; // array contains a mutable copy
strcpy(array, "OK");

char *ptr = "Can't change me"; // ptr points to some immutable
    memory
strcpy(ptr, "Will not work");
```

String literals are character arrays stored in the code segment of the program, which is immutable. Two string literals may share the same space in memory. An example follows.

```
char *str1 = "Bhuvy likes books";
char *str2 = "Bhuvy likes books";
```

The strings pointed to by str1 and str2 may actually reside in the same location in memory.

Char arrays, however, contain the literal value which has been copied from the code segment into either the stack or static memory. These following char arrays reside in different memory locations.

```
char arr1[] = "Bhuvy also likes to write";
char arr2[] = "Bhuvy also likes to write";
```

Here are some common ways to initialize a string include. Where do they reside in memory?

```
char *str = "ABC";
char str[] = "ABC";
char str[]={'A','B','C','\0'};
```

```
char ary[] = "Hello";
char *ptr = "Hello";
```

We can also print out the pointer and the contents of a C-string easily. Here is some boilerplate code to illustrate this.

```
char ary[] = "Hello";
char *ptr = "Hello";
// Print out address and contents
printf("%p : %s\n", ary, ary);
printf("%p : %s\n", ptr, ptr);
```

As mentioned before, the char array is mutable, so we can change its contents. Be careful to write within the bounds of the array. C does *not* do bounds checking at compile-time, but invalid reads/writes can get your program to crash.

```
strcpy(ary, "World"); // OK
strcpy(ptr, "World"); // NOT OK - Segmentation fault (crashes by
    default; unless SIGSEGV is blocked)
```

Unlike the array, however, we can change ptr to point to another piece of memory,

```
ptr = "World"; // OK!
ptr = ary; // OK!
ary = "World"; // NO won't compile
// ary is doomed to always refer to the original array.
printf("%p : %s\n", ptr, ptr);
strcpy(ptr, "World"); // OK because now ptr is pointing to mutable
    memory (the array)
```

Unlike pointers, that hold addresses to variables on the heap, or stack, char arrays (string literals) point to read-only memory located in the data section of the program. This means that pointers are more flexible than arrays, even though the name of an array is a pointer to its starting address.

In a more common case, pointers will point to heap memory in which case the memory referred to by the pointer **can** be modified.

Pointers

Pointers are variables that hold addresses. These addresses have a numeric value, but usually, programmers are interested in the value of the contents at that memory address. In this section, we will try to take you through a basic introduction to pointers.

Pointer Basics

Declaring a Pointer

A pointer refers to a memory address. The type of the pointer is useful – it tells the compiler how many bytes need to be read/written and delineates the semantics for pointer arithmetic (addition and subtraction).

```
int *ptr1;
char *ptr2;
```

Due to C's syntax, an int* or any pointer is not actually its own type. You have to precede each pointer variable with an asterisk. As a common gotcha, the following

```
int* ptr3, ptr4;
```

Will only declare `*ptr3` as a pointer. `ptr4` will actually be a regular int variable. To fix this declaration, ensure the `*` precedes the pointer.

```
int *ptr3, *ptr4;
```

Keep this in mind for structs as well. If one declares without a typedef, then the pointer goes after the type.

```
struct person *ptr3;
```

Reading / Writing with pointers

Let's say that `int *ptr` was declared. For the sake of discussion, let us assume that `ptr` contains the memory address `0x1000`. To write to the pointer, it must be dereferenced and assigned a value.

```
*ptr = 0; // Writes some memory.
```

What C does is take the type of the pointer which is an `int` and write `sizeof(int)` bytes from the start of the pointer, meaning that bytes `0x1000`, `0x1001`, `0x1002`, `0x1003` will all be zero. The number of bytes written depends on the pointer type. It is the same for all primitive types but structs are a little different.

Reading works roughly the same way, except you put the variable in the spot that it needs the value.

```
int double = *ptr * 2
```

Reading and writing to non-primitive types gets tricky. The compilation unit - usually the file or a header - needs to have the size of the data structure readily available. This means that opaque data structures can't be copied. Here is an example of assigning a struct pointer:

```
#include <stdio.h>

typedef struct {
  int a1;
  int a2;
} pair;

int main() {
  pair obj;
  pair zeros;
  zeros.a1 = 0;
  zeros.a2 = 0;
  pair *ptr = &obj;
  obj.a1 = 1;
  obj.a2 = 2;
  *ptr = zeros;
  printf("a1: %d, a2: %d\n", ptr->a1, ptr->a2);
  return 0;
}
```

As for reading structure pointers, don't do it directly. Instead, programmers create abstractions for creating, copying, and destroying structs. If this sounds familiar, it is what C++ originally intended to do before the standards committee went off the deep end.

Pointer Arithmetic

In addition to adding to an integer, pointers can be added to. However, the pointer type is used to determine how much to increment the pointer. A pointer is moved over by the value added times the size of the underlying type. For char pointers, this is trivial because characters are always one byte.

```
char *ptr = "Hello"; // ptr holds the memory location of ’H’
ptr += 2; // ptr now points to the first ’l’’
```

If an int is 4 bytes then ptr+1 points to 4 bytes after whatever ptr is pointing at.

```
char *ptr = "ABCDEFGH";
int *bna = (int *) ptr;
bna +=1; // Would cause iterate by one integer space (i.e 4 bytes
    on some systems)
ptr = (char *) bna;
printf("%s", ptr);
```

Notice how only ’EFGH’ is printed. Why is that? Well as mentioned above, when performing ’bna+=1’ we are increasing the **integer** pointer by 1, (translates to 4 bytes on most systems) which is equivalent to 4 characters (each character is only 1 byte) Because pointer arithmetic in C is always automatically scaled by the size of the type that is pointed to, POSIX standards forbid arithmetic on void pointers. Having said that, compilers will often treat the underlying type as char. Here is a machine translation. The following two pointer arithmetic operations are equal

```
int *ptr1 = ...;

// 1
int *offset = ptr1 + 4;

// 2
char *temp_ptr1 = (char*) ptr1;
int *offset = (int*)(temp_ptr1 + sizeof(int)*4);
```

Every time you do pointer arithmetic, take a deep breath and make sure that you are shifting over the number of bytes you think you are shifting over.

So what is a void pointer?

A void pointer is a pointer without a type. Void pointers are used when either the datatype is unknown or when interfacing C code with other programming languages without APIs. You can think of this as a raw pointer, or a memory address. malloc by default returns a void pointer that can be safely promoted to any other type.

```
void *give_me_space = malloc(10);
char *string = give_me_space;
```

C automatically promotes void* to its appropriate type. gcc and clang are not totally ISO C compliant, meaning that they will permit arithmetic on a void pointer. They will treat it as a char pointer. Do not do this because it is not portable - it is not guaranteed to work with all compilers!

Common Bugs

Nul Bytes

What's wrong with this code?

```
void mystrcpy(char*dest, char* src) {
  // void means no return value
  while( *src ) {dest = src; src ++; dest++; }
}
```

In the above code it simply changes the dest pointer to point to source string. Also the NUL bytes are not copied. Here is a better version -

```c
while( *src ) {*dest = *src; src ++; dest++; }
*dest = *src;
```

Note that it is also common to see the following kind of implementation, which does everything inside the expression test, including copying the NUL byte. However, this is bad style, as a result of doing multiple operations in the same line.

```c
while( (*dest++ = *src++ )) {};
```

Double Frees

A double free error is when a program accidentally attempt to free the same allocation twice.

```c
int *p = malloc(sizeof(int));
free(p);

*p = 123; // Oops! - Dangling pointer! Writing to memory we don't
    own anymore

free(p); // Oops! - Double free!
```

The fix is first to write correct programs! Secondly, it is a good habit to set pointers to NULL, once the memory has been freed. This ensures that the pointer cannot be used incorrectly without the program crashing.

```c
p = NULL; // No dangling pointers
```

Returning pointers to automatic variables

```c
int *f() {
  int result = 42;
  static int imok;
  return &imok; // OK - static variables are not on the stack
  return &result; // Not OK
}
```

Automatic variables are bound to stack memory only for the lifetime of the function. After the function returns, it is an error to continue to use the memory.

Insufficient memory allocation

```
struct User {
  char name[100];
};
typedef struct User user_t;

user_t *user = (user_t *) malloc(sizeof(user));
```

In the above example, we needed to allocate enough bytes for the struct. Instead, we allocated enough bytes to hold a pointer. Once we start using the user pointer we will corrupt memory. The correct code is shown below.

```
struct User {
  char name[100];
};
typedef struct User user_t;

user_t * user = (user_t *) malloc(sizeof(user_t));
```

Buffer overflow/ underflow

A famous example: Heart Bleed performed a memcpy into a buffer that was of insufficient size. A simple example: implement a strcpy and forget to add one to strlen, when determining the size of the memory required.

```
#define N (10)
int i = N, array[N];
for( ; i >= 0; i--) array[i] = i;
```

C fails to check if pointers are valid. The above example writes into array[10] which is outside the array bounds. This can cause memory corruption because that memory location is probably being used for something else. In practice, this can be harder to spot because the overflow/underflow may occur in a library call. Here is our old friend gets.

```
gets(array); // Let's hope the input is shorter than my array!
```

Strings require strlen(s)+1 bytes

Every string must have a NUL byte after the last characters. To store the string “Hi” it takes 3 bytes: [H] [i] [\0].

```
char *strdup(const char *input) {/* return a copy of 'input' */
  char *copy;
  copy = malloc(sizeof(char*));  /* nope! this allocates space for
      a pointer, not a string */
  copy = malloc(strlen(input));  /* Almost...but what about the
      null terminator? */
  copy = malloc(strlen(input) + 1); /* That's right. */
  strcpy(copy, input); /* strcpy will provide the null terminator */
  return copy;
}
```

Using uninitialized variables

```
int myfunction() {
  int x;
  int y = x + 2;
  ...
```

Automatic variables hold garbage or bit pattern that happened to be in memory or register. It is an error to assume that it will always be initialized to zero.

Assuming Uninitialized memory will be zeroed

```
void myfunct() {
  char array[10];
  char *p = malloc(10);
```

Automatic (temporary variables) and heap allocations may contain random bytes or garbage.

Logic and Program flow mistakes

These are a set of mistakes that may let the program compile but perform unintended functionality.

Equal vs. Equality

Confusingly in C, the assignment operator also returns the assigned value. Most of the time it is ignored. We can use it to initialize multiple things on the same line.

```
int p1, p2;
p1 = p2 = 0;
```

More confusingly, if we forget an equals sign in the equality operator we will end up assigning that variable. Most of the time this isn't what we want to do.

```
int answer = 3; // Will print out the answer.
if (answer = 42) {printf("The answer is %d", answer);}
```

The quick way to fix that is to get in the habit of putting constants first. This mistake is common enough in while loop conditions. Most modern compilers disallows assigning variables a condition without parenthesis.

```
  if (42 = answer) {printf("The answer is %d", answer);}
```

There are cases where we want to do it. A common example is getline.

```
while ((nread = getline(&line, &len, stream)) != -1)
```

This piece of code calls getline, and assigns the return value or the number of bytes read to nread. It also in the same line checks if that value is -1 and if so terminates the loop. It is always good practice to put parentheses around any assignment condition.

Undeclared or incorrectly prototyped functions

Some snippets of code may do the following.

```
time_t start = time();
```

The system function ‘time’ actually takes a parameter a pointer to some memory that can receive the time_t structure or NULL. The compiler fails to catch this error because the programmer omitted the valid function prototype by including `time.h`.

More confusingly this could compile, work for decades and then crash. The reason for that is that time would be found at link time, not compile-time in the C standard library which almost surely is already in memory. Since a parameter isn’t being passed, we are hoping the arguments on the stack (any garbage) is zeroed out because if it isn’t, time will try to write the result of the function to that garbage which will cause the program to SEGFAULT.

Extra Semicolons

This is a pretty simple one, don’t put semicolons when unneeded.

```
for(int i = 0; i < 5; i++) ; printf("Printed once");
while(x < 10); x++ ; // X is never incremented
```

However, the following code is perfectly OK.

```
for(int i = 0; i < 5; i++){
  printf("%d\n", i);;;;;;;;;;;;;
}
```

It is OK to have this kind of code because the C language uses semicolons (;) to separate statements. If there is no statement in between semicolons, then there is nothing to do and the compiler moves on to the next statement To save a lot of confusion, **always use braces.** It increases the number of lines of code, which is a great productivity metric.

Topics

- C-strings representation
- C-strings as pointers
- char p[]vs char* p
- Simple C string functions (strcmp, strcat, strcpy)
- sizeof char
- sizeof x vs x*
- Heap memory lifetime

- Calls to heap allocation
- Dereferencing pointers
- Address-of operator
- Pointer arithmetic
- String duplication
- String truncation
- double-free error
- String literals
- Print formatting.
- memory out of bounds errors
- static memory
- file input / output. POSIX vs. C library
- C input output: fprintf and printf
- POSIX file IO (read, write, open)
- Buffering of stdout

Questions/Exercises

- What does the following print out?

```
int main(){
fprintf(stderr, "Hello ");
fprintf(stdout, "It's a small ");
fprintf(stderr, "World\n");
fprintf(stdout, "place\n");
return 0;
}
```

- What are the differences between the following two declarations? What does `sizeof` return for one of them?

```
char str1[] = "bhuvan";
char *str2 = "another one";
```

- What is a string in C?
- Code up a simple `my_strcmp`. How about `my_strcat`, `my_strcpy`, or `my_strdup`? Bonus: Code the functions while only going through the strings *once*.
- What should each of the following lines usually return?

```c
int *ptr;
sizeof(ptr);
sizeof(*ptr);
```

- What is `malloc`? How is it different from `calloc`. Once memory is allocated how can we use `realloc`?
- What is the `&` operator? How about `*`?
- Pointer Arithmetic. Assume the following addresses. What are the following shifts?

```c
char** ptr = malloc(10); //0x100
ptr[0] = malloc(20); //0x200
ptr[1] = malloc(20); //0x300
```

  - `ptr + 2`
  - `ptr + 4`
  - `ptr[0] + 4`
  - `ptr[1] + 2000`
  - `*((int)(ptr + 1)) + 3`
- How do we prevent double free errors?
- What is the printf specifier to print a string, `int`, or `char`?
- Is the following code valid? Why? Where is `output` located?

```c
char *foo(int var){
static char output[20];
snprintf(output, 20, "%d", var);
return output;
}
```

- Write a function that accepts a path as a string, and opens that file, prints the file contents 40 bytes at a time but, every other print reverses the string (try using the POSIX API for this).
- What are some differences between the POSIX file descriptor model and C's `FILE*` (i.e. what function calls are used and which is buffered)? Does POSIX use C's `FILE*` internally or vice versa?

Rapid Fire: Pointer Arithmetic

Pointer arithmetic is important! Take a deep breath and figure out how many bytes each operation moves a pointer. The following is a rapid fire section. We'll use the following definitions:

```
int *int_; // sizeof(int) == 4;
long *long_; // sizeof(long) == 8;
char *char_;
int *short_; //sizeof(short) == 2;
int **int_ptr; // sizeof(int*) == 8;
```

How many bytes are moved over from the following additions?

1. `int_ + 1`

2. `long_ + 7`

3. `short_ - 6`

4. `short_ - sizeof(long)`

5. `long_ - sizeof(long) + sizeof(int_)`

6. `long_ - sizeof(long) / sizeof(int)`

7. `(char*)(int_ptr + sizeof(long)) + sizeof(int_)`

Rapid Fire Solutions

1. 4

2. 56

3. -12

4. -16

5. 0

6. -16

7. 72

Bibliography

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. doi: 10.1109/IEEESTD.2008.4610935.

[2] Tom Duff. Tom duff on duff's device. URL https://www.lysator.liu.se/c/duffs-device.html.

[3] Fortran 72. FORTRAN IV PROGRAMMER'S REFERENCE MANUAL. Manual, DIGITAL EQUIPMENT CORPORATION, Maynard, MASSACHUSETTS, May 1972. URL http://www.bitsavers.org/www.computer.museum.uq.edu.au/pdf/DEC-10-AFDO-D%20decsystem10%20FORTRAN%20IV%20Programmer%27s%20Reference%20Manual.pdf.

[4] Apple Inc. Xnu kernel. https://github.com/apple/darwin-xnu, 2017.

[5] ISO 1124:2005. ISO C Standard. Standard, International Organization for Standardization, Geneva, CH, March 2005. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[6] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988. ISBN 9780131103627. URL https://books.google.com/books?id=161QAAAAMAAJ.

[7] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010. ISBN 0672329468, 9780672329463.

[8] Dennis M. Ritchie. The development of the c language. *SIGPLAN Not.*, 28(3):201–208, March 1993. ISSN 0362-1340. doi: 10.1145/155360.155580. URL http://doi.acm.org/10.1145/155360.155580.

4 Processes

Who needs process isolation?

Intel Marketing on Meltdown and Spectre

To understand what a process is, you need to understand what an operating system is. An operating system is a program that provides an interface between hardware and user software as well as providing a set of tools that the software can use. The operating system manages hardware and gives user programs a uniform way of interacting with hardware as long as the operating system can be installed on that hardware. Although this idea sounds like it is the end-all, we know that there are many different operating systems with their own quirks and standards. As a solution to that, there is another layer of abstraction: POSIX or portable operating systems interface. This is a standard (or many standards now) that an operating system must implement to be POSIX compatible – most systems that we'll be studying are almost POSIX compatible due more to political reasons.

Before we talk about POSIX systems, we should understand what the idea of a kernel is generally. In an operating system (OS), there are two spaces: kernel space and user space. Kernel space is a power operating mode that allows the system to interact with the hardware and has the potential to destroy your machine. User space is where most applications run because they don't need this level of power for every operation. When a user space program needs additional power, it interacts with the hardware through a system call that is conducted by the kernel. This adds a layer of security so that normal user programs can't destroy your entire operating system. For the purposes of our class, we'll talk about single machine multiple user operating systems. This is where there is a central clock on a standard laptop or desktop. Other OSes relax the central clock requirement (distributed) or the "standardness" of the hardware (embedded systems). Other invariants make sure events happen at particular times too.

The operating system is made up of many different pieces. There may be a program running to handle incoming USB connections, another one to stay connected to the network, etc. The most important one is the kernel – although it might be a set of processes – which is the heart of the operating system. The kernel has many important tasks. The first of which is booting.

1. The computer hardware executes code from read-only memory, called firmware.
2. The firmware executes a bootloader, which often conforms to the Extensible Firmware Interface (`EFI`), which is an interface between the system firmware and the operating system.
3. The bootloader's boot manager loads the operating system kernels, based on the boot settings.
4. Your kernel executes `init` to bootstrap itself from nothing.
5. The kernel executes startup scripts like starting networking and USB handling.

6. The kernel executes userland scripts like starting a desktop, and you get to use your computer!

When a program is executing in user space, the kernel provides some important services to programs in User space.

- Scheduling processes and threads
- Handling synchronization primitives (futexes, mutexes, semaphores, etc.)
- Providing system calls such as `write` or `read`
- Managing virtual memory and low-level binary devices such as `USB` drivers
- Managing filesystems
- Handling communication over networks
- Handling communication between processes
- Dynamically linking libraries
- The list goes on and on.

The kernel creates the first process `init.d` (an alternative is system.d). *init.d* boots up programs such as graphical user interfaces, terminals, etc – by default, this is the only process explicitly created by the system. All other processes are instantiated by using the system calls `fork` and `exec` from that single process.

File Descriptors

Although these were mentioned in the last chapter, we are going to give a quick reminder about file descriptors. A zine from Julia Evans gives some more details [8].

The kernel keeps track of the file descriptors and what they point to. Later we will learn two things: that file descriptors point to more than files and that the operating system keeps track of them.

Notice that file descriptors may be reused between processes, but inside of a process, they are unique. File descriptors may have a notion of position. These are known as seekable streams. A program can read a file on disk completely because the OS keeps track of the position in the file, an attribute that belongs to your process as well.

Other file descriptors point to network sockets and various other pieces of information, that are unseekable streams.

Processes

A process is an instance of a computer program that may be running. Processes have many resources at their disposal. At the start of each program, a program gets one process, but each program can make more processes. A program consists of the following:

- A binary format: This tells the operating system about the various sections of bits in the binary – which parts are executable, which parts are constants, which libraries to include etc.
- A set of machine instructions

- A number denoting which instruction to start from
- Constants
- Libraries to link and where to fill in the address of those libraries

Processes are powerful, but they are isolated!

That means that by default, no process can communicate with another process.

This is important because in complex systems (like the University of Illinois Engineering Workstations), it is likely that different processes will have different privileges. One certainly doesn't want the average user to be able to bring down the entire system, by either purposely or accidentally modifying a process. As most of you have realized by now, if you stuck the following code snippet into a program, the variables are unshared between two parallel invocations of the program.

```
int secrets;
secrets++;
printf("%d\n", secrets);
```

On two different terminals, they would both print out 1 not 2. Even if we changed the code to attempt to affect other process instances, there would be no way to change another process' state unintentionally. However, there are other intentional ways to change the program states of other processes.

Process Contents

Memory Layout

When a process starts, it gets its own address space. Each process gets the following.

- **A Stack**. The stack is the place where automatically allocated variables and function call return addresses are stored. Every time a new variable is declared, the program moves the stack pointer down to reserve space for the variable. This segment of the stack is writable but not executable. This behavior is controlled by the no-execute (NX) bit, sometimes called the $W\hat{X}$ (write XOR execute) bit, which helps prevent malicious code, such as shellcode from being run on the stack.

  If the stack grows too far – meaning that it either grows beyond a preset boundary or intersects the heap – the program will stack overflow error, most likely resulting in a SEGFAULT. **The stack is statically allocated by default; there is only a certain amount of space to which one can write.**

- **A Heap**. The heap is a contiguous, expanding region of memory [5]. If a program wants to allocate an object whose lifetime is manually controlled or whose size cannot be determined at compile-time, it would want to create a heap variable.

  The heap starts at the top of the text segment and grows upward, meaning malloc may push the heap boundary – called the program break – upward.

  We will explore this in more depth in our chapter on memory allocation. This area is also writable but not executable. One can run out of heap memory if the system is constrained or if a program run out of addresses, a phenomenon that is more common on a 32-bit system.

- **A Data Segment**

  This segment contains two parts, an initialized data segment, and an uninitialized segment. Furthermore, the initialized data segment is divided into a readable and writable section.

  - **Initialized Data Segment** This contains all of a program's globals and any other static variables.

    This section starts at the end of the text segment and starts at a constant size because the number of globals is known at compile time. The end of the data segment is called the `program break` and can be extended via the use of brk / sbrk.

    This section is writable [10, P. 124]. Most notably, this section contains variables that were initialized with a static initializer, as follows:

    ```
    int global = 1;
    ```

  - **Uninitialized Data Segment / BSS** BSS stands for an old assembler operator known as Block Started by Symbol.

    This contains all of your globals and any other static duration variables that are implicitly zeroed out.

    Example:

    ```
    int assumed_to_be_zero;
    ```

    This variable will be zeroed; otherwise, we would have a security risk involving isolation from other processes.

    They get put in a different section to speed up process start up time.

    This section starts at the end of the data segment and is also static in size because the amount of globals is known at compile time.

    Currently, both the initialized and BSS data segments are combined and referred to as the data segment [10, P. 124], despite being somewhat different in purpose.

  - **A Text Segment**. This is where all executable instructions are stored, and is readable (function pointers) but not writable.

    The program counter moves through this segment executing instructions one after the other.

    It is important to note that this is the only executable section of the program, by default.

    If a program's code while it's running, the program most likely will SEGFAULT.

    There are ways around it, but we will not be exploring these in this course.

    Why doesn't it always start at zero? This is because of a security feature called address space layout randomization.

    The reasons for and explanation about this is outside the scope of this class, but it is good to know about its existence.

    Having said that, this address can be made constant, if a program is compiled with the DEBUG flag.

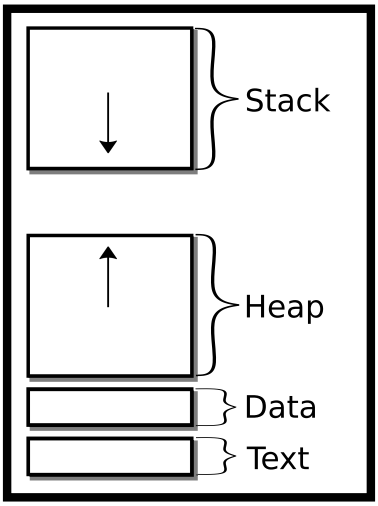


Figure 4.1: Process address space

Other Contents

To keep track of all these processes, your operating system gives each process a number called the process ID (PID). Processes are also given the PID of their parent process, called parent process ID (`PPID`). Every process has a parent, that parent could be `init.d`.

Processes could also contain the following information:

- Running State - Whether a process is getting ready, running, stopped, terminated, etc. (more on this is covered in the chapter on Scheduling).
- File Descriptors - A list of mappings from integers to real devices (files, USB flash drives, sockets)
- Permissions - What `user` the file is running on and what `group` the process belongs to. The process can then only perform operations based on the permissions given to the `user` or `group`, such as accessing files. There are tricks to make a program take a different user than who started the program i.e. `sudo` takes a program that a `user` starts and executes it as `root`. More specifically, a process has a real user ID (identifies the owner of the process), an effective user ID (used for non-privileged users trying to access files only accessible by superusers), and a saved user ID (used when privileged users perform non-privileged actions).
- Arguments - a list of strings that tell your program what parameters to run under.
- Environment Variables - a list of key-value pair strings in the form `NAME=VALUE` that one can modify. These are often used to specify paths to libraries and binaries, program configuration settings, etc.

According to the POSIX specification, a process only needs a thread and address space, but most kernel developers and users know that only these aren't enough [6].

Intro to Fork

A word of warning

Process forking is a powerful and dangerous tool. If you make a mistake resulting in a fork bomb, **you can bring down an entire system**. To reduce the chances of this, limit your maximum number of processes to a small number e.g. 40 by typing `ulimit -u 40` into a command line. Note, this limit is only for the user, which means if you fork bomb, then you won't be able to kill all created process since calling `killall` requires your shell to `fork()`. Quite unfortunate. One solution is to spawn another shell instance as another user (for example root) beforehand and kill processes from there.

Another is to use the built-in `exec` command to kill all the user processes (you only have one attempt at this).

Finally, you could reboot the system, but you only have one shot at this with the exec function.

When testing fork() code, ensure that you have either root and/or physical access to the machine involved. If you must work on fork() code remotely, remember that **kill -9 -1** will save you in the event of an emergency. Fork can be **extremely** dangerous if you aren't prepared for it. **You have been warned.**

Fork Functionality

The `fork` system call clones the current process to create a new process, called a child process. This occurs by duplicating the state of the existing process with a few minor differences.

- The child process executes the next line after the `fork()` as the parent process does.
- Just as a side remark, in older UNIX systems, the entire address space of the parent process was directly copied regardless of whether the resource was modified or not. The current behavior is for the kernel to perform a copy-on-write, which saves a lot of resources, while being time efficient [7, Copy-on-write section].

Here is a simple example:

m

```
printf("I'm printed once!\n");
fork();
// Now two processes running if fork succeeded
// and each process will print out the next line.
printf("This line twice!\n");
```

Here is a simple example of this address space cloning. The following program may print out 42 twice - but the `fork()` is after the `printf`!? Why?

m

```
#include <unistd.h> /*fork declared here*/
#include <stdio.h> /* printf declared here*/
int main() {
  int answer = 84 >> 1;
  printf("Answer: %d", answer);
  fork();
  return 0;
}
```

The printf line *is* executed only once however notice that the printed contents are not flushed to standard out. There's no newline printed, we didn't call fflush, or change the buffering mode. The output text is therefore still in process memory waiting to be sent. When fork() is executed the entire process memory is duplicated including the buffer. Thus, the child process starts with a non-empty output buffer which may be flushed when the program exits. We say may because the contents may be unwritten given a bad program exit as well.

To write code that is different for the parent and child process, check the return value of fork(). If fork() returns -1, that implies something went wrong in the process of creating a new child. One should check the value stored in *errno* to determine what kind of error occurred. Common errors include EAGAIN and ENOENT Which are essentially "try again – resource temporarily unavailable", and "no such file or directory".

Similarly, a return value of 0 indicates that we are operating in the context of the child process, whereas a positive integer shows that we are in the context of the parent process.

The positive value returned by fork() is the process id (*pid*) of the child.

A way to remember what is represented by the return value of fork is, that the child process can find its parent - the original process that was duplicated - by calling getppid() - so does not need any additional return information from fork(). However, the parent process may have many child processes, and therefore needs to be explicitly informed of its child PIDs.

According to the POSIX standard, every process only has a single parent process.

The parent process can only know the PID of the new child process from the return value of fork:

m

```
pid_t id = fork();
if (id == -1) exit(1); // fork failed
if (id > 0) {
  // Original parent
  // A child process with id 'id'
  // Use waitpid to wait for the child to finish
} else {// returned zero
  // Child Process
}
```

A slightly silly example is shown below. What will it print? Try running this program with multiple arguments.

m

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv) {
  pid_t id;
  int status;
  while (--argc && (id=fork())) {
    waitpid(id,&status,0); /* Wait for child*/
  }
  printf("%d:%s\n", argc, argv[argc]);
  return 0;
}
```

Another example is below. This is the amazing parallel apparent-O(N) *sleepsort* is today's silly winner. First published on 4chan in 2011. A version of this awful but amusing sorting algorithm is shown below. This sorting algorithm may fail to produce the correct output.

```
int main(int c, char **v) {
  while (--c > 1 && !fork());
  int val = atoi(v[c]);
  sleep(val);
  printf("%d\n", val);
  return 0;
}
```

Imagine that we ran this program like so

```
$ ./ssort 1 3 2 4
```

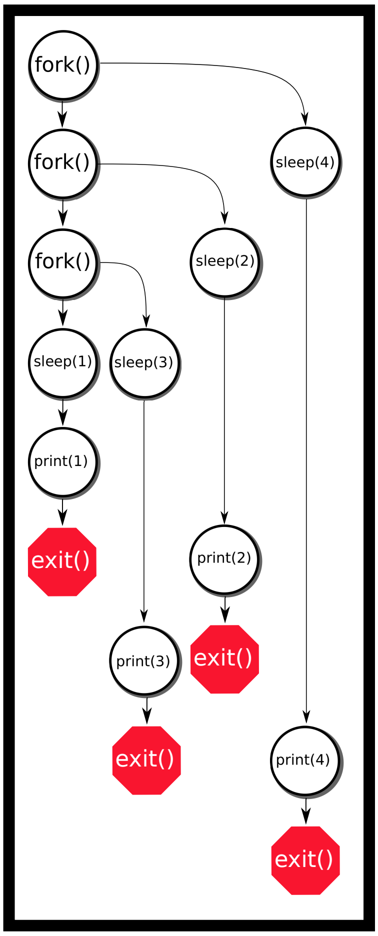


Figure 4.2: Timing of sorting 1, 3, 2, 4

The algorithm isn't actually O(N) because of how the system scheduler works. In essence, this program outsources the actual sorting to the operating system.

Fork Bomb

A 'fork bomb' is what we warned you about earlier. This occurs when there is an attempt to create an infinite number of processes. This will often bring a system to a near-standstill, as it attempts to allocate CPU time and memory to a large number of processes that are ready to run. System administrators don't like them and may set upper limits on the number of processes each user can have, or revoke login rights because they create disturbances in the Force for other users' programs. A program can limit the number of child processes created by using `setrlimit()`.

Fork bombs are not necessarily malicious - they occasionally occur due to programming errors. Below is a simple example that is malicious.

```
while (1) fork();
```

It is easy to cause one, if you are careless while calling fork, especially in a loop. Can you spot the fork bomb here?

```c
#include <unistd.h>
#define HELLO_NUMBER 10

int main(){
  pid_t children[HELLO_NUMBER];
  int i;
  for(i = 0; i < HELLO_NUMBER; i++){
    pid_t child = fork();
    if(child == -1) {
      break;
    }
    if(child == 0) {
      // Child
      execlp("ehco", "echo", "hello", NULL);
    }
    else{
      // Parent
      children[i] = child;
    }
  }

  int j;
  for(j = 0; j < i; j++){
    waitpid(children[j], NULL, 0);
  }
  return 0;
}
```

We misspelled `ehco`, so the `exec` call fails. What does this mean? Instead of creating 10 processes, we created *1024 processes, fork bombing our machine*. **How could we prevent this? Add an exit right after exec, so that if exec fails, we won't end up calling fork an unbounded number of times.** There are various other ways. What if we removed the `echo` binary? What if the binary itself creates a fork bomb?

Signals

We won't fully explore signals until the end of the course, but it is relevant to broach the subject now because various semantics related to fork and other function calls detail what a signal is.

A signal can be thought of as a software interrupt. This means that a process that receives a signal stops the execution of the current program and makes the program respond to the signal.

There are various signals defined by the operating system, two of which you may already know: SIGSEGV and SIGINT. The first is caused by an illegal memory access, and the second is sent by a user wanting to terminate a program. In each case, the program jumps from the current line being executed to the signal handler. If no signal handler is supplied by the program, a default handler is executed – such as terminating the program, or ignoring the signal.

Here is an example of a simple user-defined signal handler:

```
void handler(int signum) {
  write(1, "signaled!", 9);
  // we don't need the signum because we are only catching SIGINT
  // if you want to use the same piece of code for multiple
  // signals, check the signum
}
int main() {
  signal(SIGINT, handler);
  while(1) ;
  return 0;
}
```

A signal has four stages in its life cycle: generated, pending, blocked, and received state. These refer to when a process generates a signal, the kernel is about to deliver a signal, the signal is blocked, and when the kernel delivers a signal, each of which requires some time to complete. Read more in the introduction to the Signals chapter.

The terminology is important because fork and exec require different operations based on the state a signal is in.

To note, it is generally poor programming practice to use signals in program logic, which is to send a signal to perform a certain operation. The reason: signals have no time frame of delivery and no assurance that they will be delivered. There are better ways to communicate between two processes.

If you want to read more, feel free to skip ahead to the chapter on POSIX signals and read it over. It isn't long and gives you the long and short about how to deal with signals in processes.

POSIX Fork Details

POSIX determines the standards of fork [4]. You can read the previous citation, but do note that it can be quite verbose. Here is a summary of what is relevant:

1. Fork will return a non-negative integer on success.

2. A child will inherit any open file descriptors of the parent. That means if a parent half of the file and forks, the child will start at that offset. A read on the child's end will shift the parent's offset by the same amount. Any other flags are also carried over.

3. Pending signals are not inherited. This means that if a parent has a pending signal and creates a child, the child will not receive that signal unless another process signals the child.

4. The process will be created with one thread (more on that later. The general consensus is to not create processes and threads at the same time).

5. Since we have copy on write (COW), read-only memory addresses are shared between processes.

6. If a program sets up certain regions of memory, they can be shared between processes.
7. Signal handlers are inherited but can be changed.
8. The process' current working directory (often abbreviated to CWD) is inherited but can be changed.
9. Environment variables are inherited but can be changed.

Key differences between the parent and the child include:

- The process id returned by `getpid()`. The parent process id returned by `getppid()`.
- The parent is notified via a signal, SIGCHLD, when the child process finishes but not vice versa.
- The child does not inherit pending signals or timer alarms. For a complete list see the fork man page
- The child has its own set of environment variables.

Fork and FILEs

There are some tricky edge cases when it comes to using `FILE` and forking. First, we have to make a technical distinction. A **File Description** is the struct that a file descriptor points to. File descriptors can point to many different structs, but for our purposes, they'll point to a struct that represents a file on a filesystem. This file description contains elements like paths, how far the descriptor has read into the file, etc. A file descriptor points to a file description. This is important because when a process is forked, only the file descriptor is cloned, not the description. The following snippet contains only one description.

```
int file = open(...);
if(!fork) {
  read(file, ...);
} else {
  read(file, ...);
}
```

One process will read one part of the file, the other process will read another part of the file. In the following example, there are two descriptions caused by two different file handles.

```
if(!fork) {
  int file = open(...);
  read(file, ...);
} else {
  int file = open(...);
  read(file, ...);
}
```

Let's consider our motivating example.

```
$ cat test.txt
A
B
C
```

Take a look at this code, what does it do?

```
size_t buffer_cap = 0;
char * buffer = NULL;
ssize_t nread;
FILE * file = fopen("test.txt", "r");
int count = 0;
while((nread = getline(&buffer, &buffer_cap, file) != -1) {
  printf("%s", buffer);
  if(fork() == 0) {
    exit(0);
  }
  wait(NULL);
}
```

The initial thought may be that it prints the file line by line with some extra forking. It is actually undefined behavior because we didn't prepare the file descriptors. To make a long story short, here is what to do to avoid the example.

1. You as the programmer need to make sure that all of your file descriptors are prepared before forking.

2. If it is a file descriptor or an unbuffered FILE*, it is already prepared.

3. If the FILE* is open for reading and has been read fully, it is already prepared.

4. Otherwise, the FILE* **must** be fflush'ed or closed to be prepared.

5. If the file descriptor is prepared, it must unactive in the parent process if the child process is using it or vice versa. A process is using it if it is read or written or if that process *for whatever reason* calls exit. If a process uses it when the other process is as well, the whole application's behavior is undefined.

So how would we fix the code? We would have to flush the file before forking and refrain from using it until after the wait call – more on the specifics of this next section.

```
size_t buffer_cap = 0;
char * buffer = NULL;
ssize_t nread;
FILE * file = fopen("test.txt", "r");
int count = 0;
while((nread = getline(&buffer, &buffer_cap, file) != -1) {
  printf("%s", buffer);
```

```c
  fflush(file);
  if(fork() == 0) {
    exit(0);
  }
  wait(NULL);
}
```

What if the parent process and the child process need to perform asynchronously and need to keep the file handle open? Due to event ordering, we need to make sure that parent process knows that the child is finished using `wait`. We'll talk about Inter-Process communication in a later chapter, but now we can use the double fork method.

```c
//...
fflush(file);
pid_t child = fork();
if(child == 0) {
  fclose(file);
  if (fork() == 0) {
    // Do asynchronous work
    // Safe exit, this child doesn't know about
    // the file descriptor
    exit(0);
  }
  exit(0);
}
waitpid(child, NULL, 0);
```

Extra: Explanation of the Fork-FILE Problem

To parse the POSIX documentation, we'll have to go deep into the terminology. The sentence that sets the expectation is the following

> The result of function calls involving any one handle (the "active handle") is defined elsewhere in this volume of POSIX.1-2008, but if two or more handles are used, and any one of them is a stream, the application shall ensure that their actions are coordinated as described below. If this is not done, the result is undefined.

What this means is that if we don't follow POSIX to the letter when using two file descriptors that refer to the same description across processes, we get undefined behavior. To be technical, the file descriptor must have a "position" meaning that it needs to have a beginning and an end like a file, not like an arbitrary stream of bytes. POSIX then goes on to introduce the idea of an active handle, where a handle may be a file descriptor or a `FILE*` pointer. File handles don't have a flag called "active". An active file descriptor is one that is currently being used for reading and writing and other operations (such as `exit`). The standard says that before a `fork` that the *application* or your code must execute a series of steps to prepare the state of the file. In simplified terms, the descriptor needs to be closed, flushed, or read to its entirety – the gory details are explained later.

> For a handle to become the active handle, the application shall ensure that the actions below are performed between the last use of the handle (the current active handle) and the first use of the second handle (the future active handle). The second handle then becomes the active handle. All activity by the application affecting the file offset on the first handle shall be suspended until it again becomes the active file handle. (If a stream function has as an underlying function one that affects the file offset, the stream function shall be considered to affect the file offset.)

Summarizing as if two file descriptors are actively being used, the behavior is undefined. The other note is that after a fork, the library code must prepare the file descriptor as if the other process were to make the file active at any time. The last bullet point concerns itself with how a process prepares a file descriptor in our case.

> If the stream is open with a mode that allows reading and the underlying open file description refers to a device that is capable of seeking, the application shall either perform an fflush(), or the stream shall be closed.

The documentation says that the child needs to perform an fflush or close the stream because the file descriptor needs to be prepared in case the parent process needs to make it active. glibc is in a no-win situation if it closes a file descriptor that the parent may expect to be open, so it'll opt for the fflush on exit because exit in POSIX terminology counts as accessing a file. That means that for our parent process, this clause gets triggered.

> If any previous active handle has been used by a function that explicitly changed the file offset, except as required above for the first handle, the application shall perform an lseek() or fseek() (as appropriate to the type of handle) to an appropriate location.

Since the child calls fflush and the parent didn't prepare, the operating system chooses to where the file gets reset. Different file systems will do different things which are supported by the standard. The OS may look at modification times and conclude that the file hasn't changed so no resets are needed or may conclude that exit denotes a change and needs to rewind the file back to the beginning.

Waiting and Executing

If the parent process wants to wait for the child to finish, it must use `waitpid` (or `wait`), both of which wait for a child to change process states, which can be one of the following:

1. The child terminated
2. The child was stopped by a signal
3. The child was resumed by a signal

Note that waitpid can be set to be non-blocking, which means they will return immediately, letting a program know if the child has exited.

```
pid_t child_id = fork();
```

```
if (child_id == -1) {perror("fork"); exit(EXIT_FAILURE);}
if (child_id > 0) {
  // We have a child! Get their exit code
  int status;
  waitpid( child_id, &status, 0 );
  // code not shown to get exit status from child
} else {// In child ...
  // start calculation
  exit(123);
}
```

`wait` is a simpler version of `waitpid`. `wait` accepts a pointer to an integer and waits on any child process. After the first one changes state, `wait` returns. Here is the behavior of `waitpid`:

1. A program *can* wait on a specific process, or it can pass in special values for the `pid` to do different things (check the man pages).
2. The last parameter to waitpid is an option parameter. The options are listed below:
3. WNOHANG - Return whetherthe searched process has exited
4. WNOWAIT - Wait, but leave the child wait-able by another wait call
5. WEXITED - Wait for exited children
6. WSTOPPED - Wait for stopped children
7. WCONTINUED - Wait for continued children

Exit statuses or the value stored in the integer pointer for both of the calls above are explained below.

Exit statuses

To find the return value of `main()` or value included in `exit()`), Use the `Wait macros` - typically a program will use `WIFEXITED` and `WEXITSTATUS` . See `wait`/`waitpid` man page for more information.

```
int status;
pid_t child = fork();
if (child == -1) {
  return 1; //Failed
}
if (child > 0) {
  // Parent, wait for child to finish
  pid_t pid = waitpid(child, &status, 0);
  if (pid != -1 && WIFEXITED(status)) {
    int exit_status = WEXITSTATUS(status);
    printf("Process %d returned %d" , pid, exit_status);
  }
} else {
  // Child, do something interesting
  execl("/bin/ls", "/bin/ls", ".", (char *) NULL); // "ls ."
}
```

A process can only have 256 return values, the rest of the bits are informational, and the information is extracted with bit shifting. However, the kernel has an internal way of keeping track of signaled, exited, or stopped processes. This API is abstracted so that that the kernel developers are free to change it at will. Remember: these macros only make sense if the precondition is met. For example, a process' exit status won't be defined if the process isn't signaled. The macros will not do the checking for the program, so it's up to the programmer to make sure the logic is correct. As an example above, the program should use the `WIFSTOPPED` to check if a process was stopped and then the `WSTOPSIG` to find the signal that stopped it. As such, there is no need to memorize the following. This is a high-level overview of how information is stored inside the status variables. From the `sys/wait.h` of an old Berkeley Standard Distribution(BSD) kernel [1]:

```
/* If WIFEXITED(STATUS), the low-order 8 bits of the status. */
#define _WSTATUS(x) (_W_INT(x) & 0177)
#define _WSTOPPED 0177  /* _WSTATUS if process is stopped */
#define WIFSTOPPED(x) (_WSTATUS(x) == _WSTOPPED)
#define WSTOPSIG(x) (_W_INT(x) >> 8)
#define WIFSIGNALED(x) (_WSTATUS(x) != _WSTOPPED && _WSTATUS(x) !=
    0)
#define WTERMSIG(x) (_WSTATUS(x))
#define WIFEXITED(x) (_WSTATUS(x) == 0)
```

There is a convention about exit codes. If the process exited normally and everything was successful, then a zero should be returned. Beyond that, there aren't too many widely accepted conventions. If a program specifies return codes to mean certain conditions, it may be able to make more sense of the 256 error codes. For example, a program could return `1` if the program went to stage 1 (like writing to a file) `2` if it did something else, etc. Usually, UNIX programs are not designed to follow this policy, for the sake of simplicity.

Zombies and Orphans

It is good practice to wait on your process' children. If a parent doesn't wait on your children they become, what are called zombies. Zombies are created when a child terminates and then takes up a spot in the kernel process table for your process. The process table keeps track of the following information about a process: PID, status, and how it was killed. The only way to get rid of a zombie is to wait on your children. If a long-running parent never waits for your children, it may lose the ability to fork.

Having said that, a program doesn't always need to wait for your children! Your parent process can continue to execute code without having to wait for the child process. If a parent dies without waiting on its children, a process can orphan its children. Once a parent process completes, any of its children will be assigned to `init` - the first process, whose PID is 1. Therefore, these children would see `getppid()` return a value of 1. These orphans will eventually finish and for a brief moment become a zombie. The init process automatically waits for all of its children, thus removing these zombies from the system.

Extra: Asynchronously Waiting

Warning: This section uses signals which are partially introduced. The parent gets the signal SIGCHLD when a child completes, so the signal handler can wait for the process. A slightly simplified version is shown below.

```
pid_t child;

void cleanup(int signal) {
  int status;
  waitpid(child, &status, 0);
  write(1,"cleanup!\n",9);
}
int main() {
  // Register signal handler BEFORE the child can finish
  signal(SIGCHLD, cleanup); // or better - sigaction
  child = fork();
  if (child == -1) {exit(EXIT_FAILURE);}

  if (child == 0) {
    // Do background stuff e.g. call exec
  } else {/* I'm the parent! */
    sleep(4); // so we can see the cleanup
    puts("Parent is done");
  }
  return 0;
}
```

However, the above example misses a couple of subtle points.

1. More than one child may have finished but the parent will only get one SIGCHLD signal (signals are not queued)
2. SIGCHLD signals can be sent for other reasons (e.g. a child process has temporarily stopped)
3. It uses the deprecated signal code, instead of the more portable sigaction.

A more robust code to reap zombies is shown below.

```
void cleanup(int signal) {
  int status;
  while (waitpid((pid_t) (-1), 0, WNOHANG) > 0) {

  }
}
```

exec

To make the child process execute another program, use one of the `exec` functions after forking. The `exec` set of functions replaces the process image with that of the specified program. This means that any lines of code after the `exec` call are replaced with those of the executed program. Any other work a program wants the child process to do should be done before the `exec` call. The naming schemes can be shortened mnemonically.

1. e – An array of pointers to environment variables is explicitly passed to the new process image.
2. l – Command-line arguments are passed individually (a list) to the function.
3. p – Uses the PATH environment variable to find the file named in the file argument to be executed.
4. v – Command-line arguments are passed to the function as an array (vector) of pointers.

Note that if the information is passed via an array, the last element must be followed by a NULL element to terminate the array.

An example of this code is below. This code executes `ls`

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char**argv) {
  pid_t child = fork();
  if (child == -1) return EXIT_FAILURE;
  if (child) {
    int status;
    waitpid(child , &status ,0);
    return EXIT_SUCCESS;

  } else {
    // Other versions of exec pass in arguments as arrays
    // Remember first arg is the program name
    // Last arg must be a char pointer to NULL

    execl("/bin/ls", "/bin/ls", "-alh", (char *) NULL);

    // If we get to this line, something went wrong!
    perror("exec failed!");
  }
}
```

Try to decode the following example

```
#include <unistd.h>
#include <fcntl.h> // O_CREAT, O_APPEND etc. defined here

int main() {
  close(1); // close standard out
  open("log.txt", O_RDWR | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR);
  puts("Captain's log");
  chdir("/usr/include");
  // execl( executable, arguments for executable including program
      name and NULL at the end)

  execl("/bin/ls", /* Remaining items sent to ls*/ "/bin/ls", ".",
      (char *) NULL); // "ls ."
  perror("exec failed");
  return 0;
}
```

The example writes "Captain's Log" to a file then prints everything in /usr/include to the same file. There's no error checking in the above code (we assume close, open, chdir etc. work as expected).

1. open – will use the lowest available file descriptor (i.e. 1) ; so standard out(stdout) is now redirected to the log file.

2. chdir – Change the current directory to /usr/include

3. execl – Replace the program image with /bin/ls and call its main() method

4. perror – We don't expect to get here - if we did then exec failed.

5. We need the "return 0;" because compilers complain if we don't have it.

POSIX Exec Details

POSIX details all of the semantics that exec needs to cover [3]. Note the following

1. File descriptors are preserved after an exec. That means if a program open a file and doesn't to close it, it remains open in the child. This is a problem because usually the child doesn't know about those file descriptors. Nevertheless, they take up a slot in the file descriptor table and could possibly prevent other processes from accessing the file. The one exception to this is if the file descriptor has the Close-On-Exec flag set (O_CLOEXEC) – we will go over setting flags later.

2. Various signal semantics. The executed processes preserve the signal mask and the pending signal set but does not preserve the signal handlers since it is a different program.

3. Environment variables are preserved unless using an environ version of exec

4. The operating system may open up 0, 1, 2 – stdin, stdout, stderr, if they are closed after exec, most of the time they leave them closed.

5. The executed process runs as the same PID and has the same parent and process group as the previous process.

6. The executed process is run on the same user and group with the same working directory

Shortcuts

system pre-packs the above code [9, P. 371]. The following is a snippet of how to use system.

```
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char**argv) {
  system("ls"); // execl("/bin/sh", "/bin/sh", "-c", "\\"ls\\"")
  return 0;
}
```

The system call will fork, execute the command passed by parameter and the original parent process will wait for this to finish. This also means that system is a blocking call. The parent process can't continue until the process started by system exits. Also, system actually creates a shell that is then given the string, which is more overhead than using exec directly. The standard shell will use the PATH environment variable to search for a filename that matches the command. Using system will usually be sufficient for many simple run-this-command problems but can quickly become limiting for more complex or subtle problems, and it hides the mechanics of the fork-exec-wait pattern, so we encourage you to learn and use fork exec and waitpid instead. It also tends to be a huge security risk. By allowing someone to access a shell version of the environment, the program can run into all sorts of problems:

```
int main(int argc, char**argv) {
  char *to_exec = asprintf("ls %s", argv[1]);
  system(to_exec);
}
```

Passing something along the lines of argv[1] = "; sudo su" is a huge security risk called privilege escalation.

The fork-exec-wait Pattern

A common programming pattern is to call fork followed by exec and wait. The original process calls fork, which creates a child process. The child process then uses exec to start the execution of a new program. Meanwhile, the parent uses wait (or waitpid) to wait for the child process to finish.

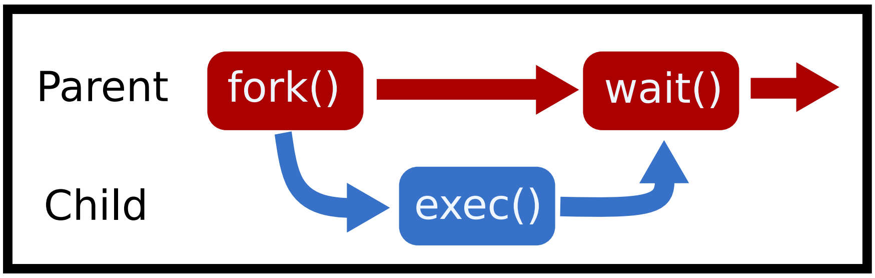


Figure 4.3: Fork, exec, wait diagram

```
#include <unistd.h>

int main() {
  pid_t pid = fork();
  if (pid < 0) {// fork failure
    exit(1);
  } else if (pid > 0) {
    int status;
    waitpid(pid, &status, 0);
  } else {
    execl("/bin/ls", "/bin/ls", NULL);
    exit(1); // For safety.
  }
}
```

Why not execute ls directly? The reason is that now we have a monitor program – our parent that can do other things. It can proceed and execute another function, or it can also modify the state of the system or read the output of the function call.

Environment Variables

Environment variables are variables that the system keeps for all processes to use. Your system has these set up right now! In Bash, some are already defined

```
$ echo $HOME
/home/bhuvy
$ echo $PATH
/usr/local/sbin:/usr/bin:...
```

How would a program later these in C? They can call getenv and setenv function respectively.

```
char* home = getenv("HOME"); // Will return /home/bhuvy
setenv("HOME", "/home/bhuvan", 1 /*set overwrite to true*/ );
```

Environment variables are important because they are inherited between processes and can be used to specify a standard set of behaviors [2], although you don't need to memorize the options. Another security related concern is that environment variables cannot be read by an outside process, whereas argv can be.

Further Reading

Read the man pages and the POSIX groups above! Here are some guiding questions. Note that we aren't expecting you to memorize the man page.

- What is one reason fork may fail?
- Does fork copy all pages to the child?
- Are file descriptors cloned between parent and child?
- Are file descript**ions** cloned between parent and child?
- What is the difference between exec calls ending in an e?
- What is the difference between l and v in an exec call? How about p?
- When does exec error? What happens?
- Does wait only notify if a child has exited?
- Is it an error to pass a negative value into wait?
- How does one extract information out of the status?
- Why may wait fail?
- What happens when a parent doesn't wait on their children?
- fork
- exec
- wait

Topics

- Correct use of fork, exec and waitpid
- Using exec with a path
- Understanding what fork and exec and waitpid do. E.g. how to use their return values.
- SIGKILL vs SIGSTOP vs SIGINT.

- What signal is sent when press CTRL-C at a terminal?
- Using kill from the shell or the kill POSIX call.
- Process memory isolation.
- Process memory layout (where is the heap, stack etc; invalid memory addresses).
- What is a fork bomb, zombie and orphan? How to create/remove them.
- getpid vs getppid
- How to use the WAIT exit status macros WIFEXITED etc.

Questions/Exercises

- What is the difference between execs with a p and without a p? What does the operating system
- How does a program pass in command line arguments to `execl*`? How about `execv*`? What should be the first command line argument by convention?
- How does a program know if `exec` or `fork` failed?
- What is the `int *status` pointer passed into wait? When does wait fail?
- What are some differences between `SIGKILL`, `SIGSTOP`, `SIGCONT`, `SIGINT`? What are the default behaviors? Which ones can a program set up a signal handler for?
- What signal is sent when you press `CTRL-C`?
- My terminal is anchored to PID = 1337 and has become unresponsive. Write me the terminal command and the C code to send `SIGQUIT` to it.
- Can one process alter another processes memory through normal means? Why?
- Where is the heap, stack, data, and text segment? Which segments can a program write to? What are invalid memory addresses?
- Code up a fork bomb in C (please don't run it).
- What is an orphan? How does it become a zombie? What should a parent do to avoid this?
- Don't you hate it when your parents tell you that you can't do something? Write a program that sends `SIGSTOP` to a parent process.
- Write a function that fork exec waits an executable, and using the wait macros tells me if the process exited normally or if it was signaled. If the process exited normally, then print that with the return value. If not, then print the signal number that caused the process to terminate.

Bibliography

[1] Source to sys/wait.h. URL http://unix.superglobalmegacorp.com/Net2/newsrc/sys/wait.h.html.

[2] Environment variables, Jul 2018. URL https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap08.html.

[3] exec, Jul 2018. URL https://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html.

[4] fork, Jul 2018. URL https://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html.

[5] Overview of malloc, Mar 2018. URL https://sourceware.org/glibc/wiki/MallocInternals.

[6] Definitions, Jul 2018. URL http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html#tag_03_210.

[7] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN 0596005652.

[8] Julia Evans. File descriptors, Apr 2018. URL https://drawings.jvns.ca/file-descriptors/.

[9] Larry Jones. Wg14 n1539 committee draft iso/iec 9899: 201x, 2010.

[10] Peter Van der Linden. *Expert C programming: deep C secrets*. Prentice Hall Professional, 1994.

5

Memory Allocators

Memory memory everywhere but not an allocation to be made

A fragmented heap

Introduction

Memory allocation is important! Allocating and deallocating heap memory is one of the most common operations in any application. The heap at the system level is contiguous series of addresses that the program can expand or contract and use as its own accord [2]. In POSIX, this is called the system break. We use `sbrk` to move the system break. Most programs don't interact directly with this call, they use a memory allocation system around it to handle chunking up and keeping track of which memory is allocated and which is freed.

We will mainly be looking into simple allocators. Just know that there are other ways of dividing up memory like with `mmap` or other allocation schemes and methods like `jemalloc`.

C Memory Allocation API

- `malloc(size_t bytes)` is a C library call and is used to reserve a contiguous block of memory that may be uninitialized [4, P. 348]. Unlike stack memory, the memory remains allocated until `free` is called with the same pointer. If `malloc` can either return a pointer to at least that much free space requested or `NULL`. That means that malloc can return NULL even if there is some spaces. Robust programs should check the return value. If your code assumes `malloc` succeeds, and it does not, then your program will likely crash (segfault) when it tries to write to address 0. Also, malloc leave garbage in memory because of performance – check your code to make sure that a program all program values are initialized.
- `realloc(void *space, size_t bytes)` allows a program to resize an existing memory allocation that was previously allocated on the heap (via malloc, calloc, or realloc) [4, P. 349]. The most common use of realloc is to resize memory used to hold an array of values. There are two gotchas with realloc. One, a new pointer may be returned. Two, it can fail. A naive but readable version of realloc is suggested below with sample usage.

```
void * realloc(void * ptr, size_t newsize) {
  // Simple implementation always reserves more memory
  // and has no error checking
  void *result = malloc(newsize);
  size_t oldsize = ... //(depends on allocator’s internal data structure)
  if (ptr) memcpy(result, ptr, newsize < oldsize ? newsize : oldsize);
  free(ptr);
  return result;
}

int main() {
  // 1
  int *array = malloc(sizeof(int) * 2);
  array[0] = 10; array[1] = 20;
  // Oops need a bigger array - so use realloc..
  array = realloc(array, 3 * sizeof(int));
  array[2] = 30;

}
```

The above code is fragile. If realloc fails then the program leaks memory. Robust code checks for the return value and only reassigns the original pointer if not NULL.

```
int main() {
  // 1
  int *array = malloc(sizeof(int) * 2);
  array[0] = 10; array[1] = 20;
  void *tmp = realloc(array, 3 * sizeof(int));
  if (tmp == NULL) {
    // Nothing to do here.
  } else if (tmp == array) {
    // realloc returned same space
    array[2] = 30;
  } else {
    // realloc returned different space
    array = tmp;
    array[2] = 30;
  }

}
```

- calloc(size_t nmemb, size_t size) initializes memory contents to zero and also takes two arguments: the number of items and the size in bytes of each item. An advanced discussion of these limitations is in this article. Programmers often use calloc rather than explicitly calling memset after malloc, to set the memory contents to zero because certain performance considerations are taken into account. Note calloc(x,y) is identical to calloc(y,x), but you should follow the conventions of the manual. A naive implementation of calloc is below.

```
void *calloc(size_t n, size_t size) {
  size_t total = n * size; // Does not check for overflow!
  void *result = malloc(total);

  if (!result) return NULL;

  // If we're using new memory pages
  // allocated from the system by calling sbrk
  // then they will be zero so zero-ing out is unnecessary,
  // We will be non-robust and memset either way.
  return memset(result, 0, total);
}
```

- free takes a pointer to the start of a piece of memory and makes it available for use in subsequent calls to the other allocation functions. This is important because we don't want every process in our address space to take an enormous amount of memory. Once we are done using memory, we stop using it with free. A simple usage is below.

```
int *ptr = malloc(sizeof(*ptr));
do_something(ptr);
free(ptr);
```

  If a program uses a piece of memory after it is freed - that is undefined behavior.

Heaps and sbrk

The heap is part of the process memory and varies in size. Heap memory allocation is performed by the C library when a program calls malloc (calloc, realloc) and free. By calling sbrk the C library can increase the size of the heap as your program demands more heap memory. As the heap and stack need to grow, we put them at opposite ends of the address space. Stacks don't grow like a heap, new parts of the stack are allocated for new threads. For typical architectures, the heap will grow upwards and the stack grows downwards.

Nowadays, Modern operating system memory allocators no longer need sbrk. Instead, they can request independent regions of virtual memory and maintain multiple memory regions. For example, gibibyte requests may be placed in a different memory region than small allocation requests. However, this detail is an unwanted complexity.

Programs don't need to call brk or sbrk typically, though calling sbrk(0) can be interesting because it tells a program where your heap currently ends. Instead programs use malloc, calloc, realloc and free which are part of the C library. The internal implementation of these functions may call sbrk when additional heap memory is required.

```
void *top_of_heap = sbrk(0);
malloc(16384);
void *top_of_heap2 = sbrk(0);
printf("The top of heap went from %p to %p \n", top_of_heap,
    top_of_heap2);
// Example output: The top of heap went from 0x4000 to 0xa000
```

Note that the memory that was newly obtained by the operating system must be zeroed out. If the operating system left the contents of physical RAM as-is, it might be possible for one process to learn about the memory of another process that had previously used the memory. This would be a security leak. Unfortunately, this means that for malloc requests before any memory has been freed is *often* zero. This is unfortunate because many programmers mistakenly write C programs that assume allocated memory will *always* be zero.

```
char* ptr = malloc(300);
// contents is probably zero because we get brand new memory
// so beginner programs appear to work!
// strcpy(ptr, "Some data"); // work with the data
free(ptr);
// later
char *ptr2 = malloc(300); // Contents might now contain existing
    data and is probably not zero
```

Intro to Allocating

Let's try to write Malloc. Here is our first attempt at it – the naive version.

```
void* malloc(size_t size)
{
    // Ask the system for more bytes by extending the heap space.
    // sbrk returns -1 on failure
    void *p = sbrk(size);
    if(p == (void *) -1) return NULL; // No space left
    return p;
}
  void free() {/* Do nothing */}
```

Above is the simplest implementation of malloc, there are a few drawbacks though.

- System calls are slow compared to library calls. We should reserve a large amount of memory and only occasionally ask for more from the system.
- No reuse of freed memory. Our program never re-uses heap memory - it keeps asking for a bigger heap.

If this allocator was used in a typical program, the process would quickly exhaust all available memory. Instead, we need an allocator that can efficiently use heap space and only ask for more memory when necessary. Some programs use this type of allocator. Consider a video game allocating objects to load the next scene. It is considerably faster to do the above and throw the entire block of memory away than it is to do the following placement strategies.

Placement Strategies

During program execution, memory is allocated and deallocated, so there will be a gap in the heap memory that can be re-used for future memory requests. The memory allocator needs to keep track of which parts of the heap are currently allocated and which are parts are available. Suppose our current heap size is 64K. Let's say that our heap looks like the following table.

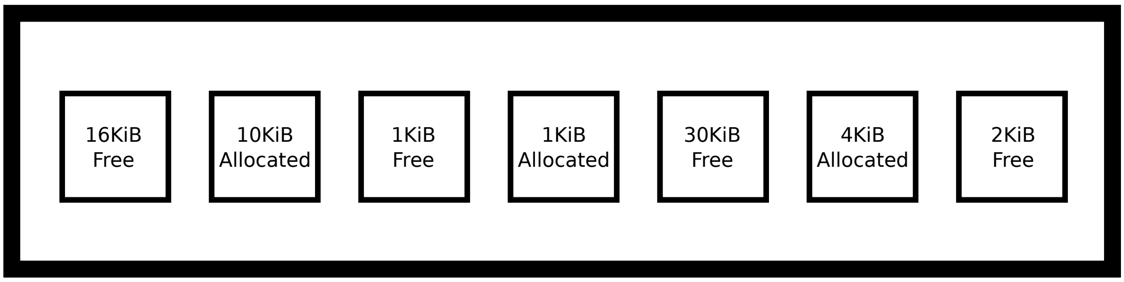


Figure 5.1: Empty heap blocks

If a new malloc request for 2KiB is executed (`malloc(2048)`), where should `malloc` reserve the memory? It could use the last 2KiB hole, which happens to be the perfect size! Or it could split one of the other two free holes. These choices represent different placement strategies. Whichever hole is chosen, the allocator will need to split the hole into two. The newly allocated space, which will be returned to the program and a smaller hole if there is spare space left over. A perfect-fit strategy finds the smallest hole that is of sufficient size (at least 2KiB):

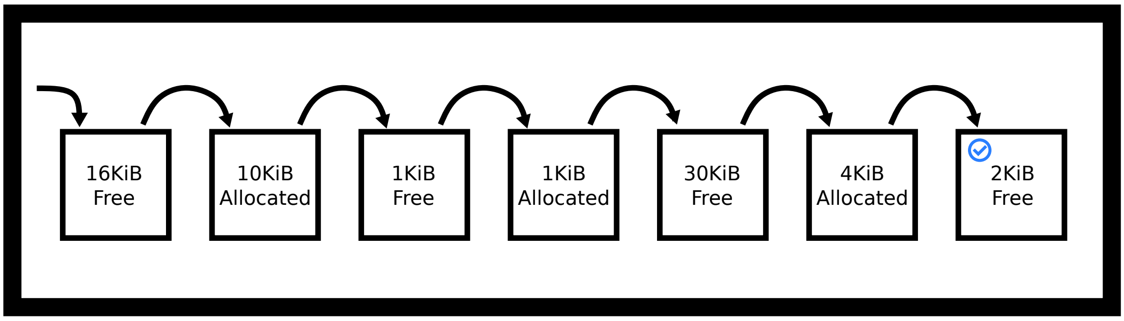


Figure 5.2: Best fit finds an exact match

A worst-fit strategy finds the largest hole that is of sufficient size so break the 30KiB hole into two:

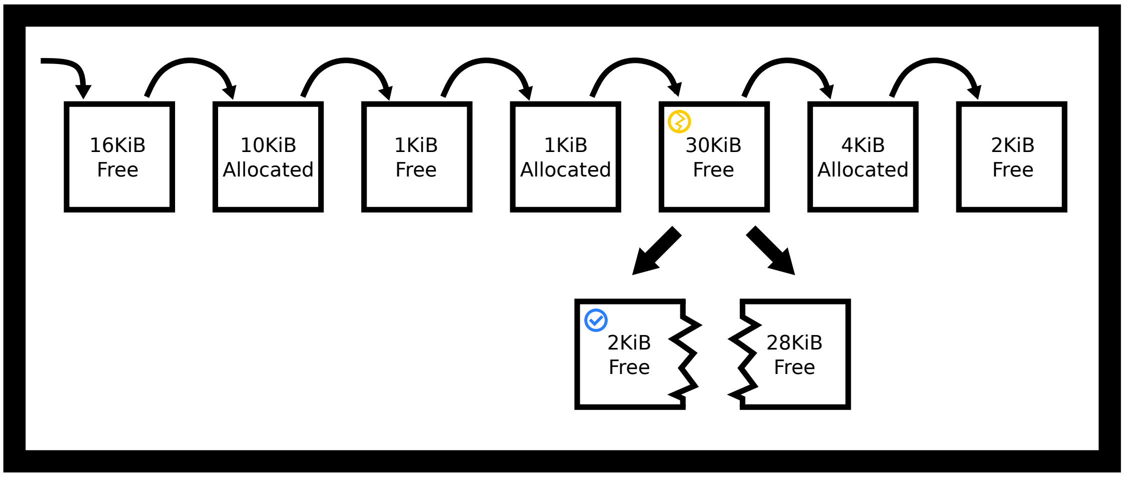


Figure 5.3: Worst fit finds the worst match

A first-fit strategy finds the first available hole that is of sufficient size so break the 16KiB hole into two. We don't even have to look through the entire heap!

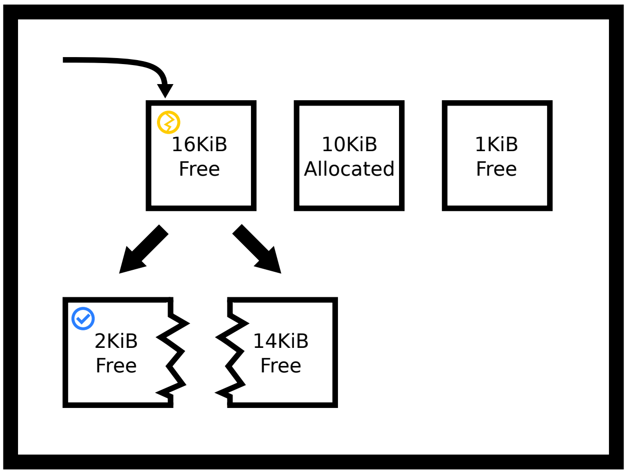


Figure 5.4: First fit finds the first match

One thing to keep in mind is those placement strategies don't need to replace the block. For example, our first fit allocator could've returned the original block unbroken. Notice that this would lead to about 14KiB of space to be unused by the user and the allocator. We call this internal fragmentation.

In contrast, external fragmentation is that even though we have enough memory in the heap, it may be divided up in a way so a continuous block of that size is unavailable. In our previous example, of the 64KiB of heap memory, 17KiB is allocated, and 47KiB is free. However, the largest available block is only 30KiB because our available unallocated heap memory is fragmented into smaller pieces.

Placement Strategy Pros and Cons

The challenges of writing a heap allocator are

- Need to minimize fragmentation (i.e. maximize memory utilization)
- Need high performance
- Fiddly implementation – lots of pointer manipulation using linked lists and pointer arithmetic.
- Both fragmentation and performance depend on the application allocation profile, which can be evaluated but not predicted and in practice, under-specific usage conditions, a special-purpose allocator can often out-perform a general-purpose implementation.
- The allocator doesn't know the program's memory allocation requests in advance. Even if we did, this is the Knapsack problem which is known to be NP-hard!

Different strategies affect the fragmentation of heap memory in non-obvious ways, which only are discovered by mathematical analysis or careful simulations under real-world conditions (for example simulating the memory allocation requests of a database or webserver).

First, we will have a more mathematical, one-shot approach to each of these algorithms [3]. The paper describes a scenario where you have a certain number of bins, and a certain number of allocations, and you are trying to fit the allocations in as few bins as possible, hence using as little memory as possible. The paper discusses theoretical implications and puts a nice limit on the ratios in the long run between the ideal memory usage and the actual memory usage. For those who are interested, the paper concludes that actual memory usage over ideal memory usage as the number of bins increases – the bins can have any distribution – is about 1.7 for first fit and lower bounded by 1.7 for best fit. The problem with this analysis is that few real-world applications need this type of one-shot allocation. Video game object allocations will typically designate a different subheap for each level and fill up that subheap if they need a quick memory allocation scheme that they can throw away.

In practice, we'll be using the result from a more rigorous survey conducted in 2005 [7]. The survey makes sure to note that memory allocation is a moving target. A good allocation scheme to one program may not be a good allocation scheme for another program. Programs don't uniformly follow the distribution of allocations. The survey talks about all the allocation schemes that we have introduced as well as a few extra ones. Here are some summarized takeaways

1. Best fit may have problems when a block is chosen that is almost the right size, and the remaining space is split so small that a program probably won't use it. A way to get around this could be to set a threshold for splitting. This small splitting isn't observed as frequently under a regular workload. Also, the worst-case behavior of best fit is bad, but it doesn't usually happen [p. 43].
2. The survey also talks about an important distinction of first fit. There are multiple notions of first. First could be ordered in terms of the time of freeing, or it could be ordered through the addresses of the start of the block, or it could be ordered by the time of last free – first being least recently used. The survey didn't go too in-depth into the performance of each but did make a note that address ordered and least recently used lists ended up with better performance than the most recently used first.
3. The survey concludes by first saying that under simulated random (assuming uniform at random) workloads, best fit and first fit do as well. Even in practice, both best and address ordered first fit do about as equally as well with a splitting threshold and coalescing. The reasons why aren't entirely known.

Some additional notes we make

1. Best fit may take less time than a full heap scan. When a block of perfect size or of perfect size within a threshold is found, that can be returned, depending on what edge-case policy you have.
2. Worst fit follows this as well. Your heap could be represented with the max-heap data structure and each allocation call could simply pop the top off, reheapify, and possibly insert a split memory block. Using Fibonacci heaps, however, could be extremely inefficient.

3. First fit needs to have a block ordering. Most of the time programmers will default to linked lists which is a fine choice. There aren't too many improvements you can make with a least recently used and most recently used linked list policy, but with address ordered linked lists you can speed up insertion from O(n) to O(log(n)) by using a randomized skip-list in conjunction with your singly-linked list. An insert would use the skip list as shortcuts to find the right place to insert the block and removal would go through the list as normal.

4. There are many placement strategies that we haven't talked about, one is next-fit which is first fit on the next fit block. This adds a deterministic randomness – pardon the oxymoron. You won't be expected to know this algorithm, know as you are implementing a memory allocator as part of a machine problem, there are more than these.

Memory Allocator Tutorial

A memory allocator needs to keep track of which bytes are currently allocated and which are available for use. This section introduces the implementation and conceptual details of building an allocator, or the actual code that implements `malloc` and `free`.

Conceptually, we are thinking about creating linked lists and lists of blocks! Please enjoy the following ASCII art. bt is short for boundary tag.

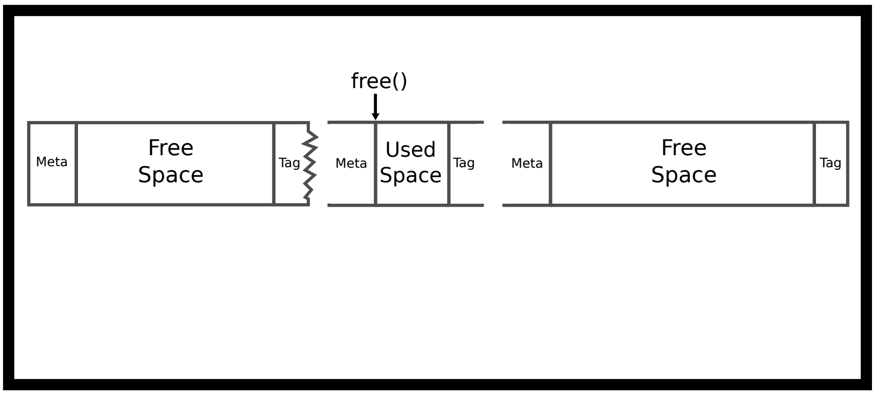


Figure 5.5: 3 Adjacent Memory blocks

We will have implicit pointers in our next block, meaning that we can get from one block to another using addition. This is in contrast to an explicit `metadata *next` field in our meta block.

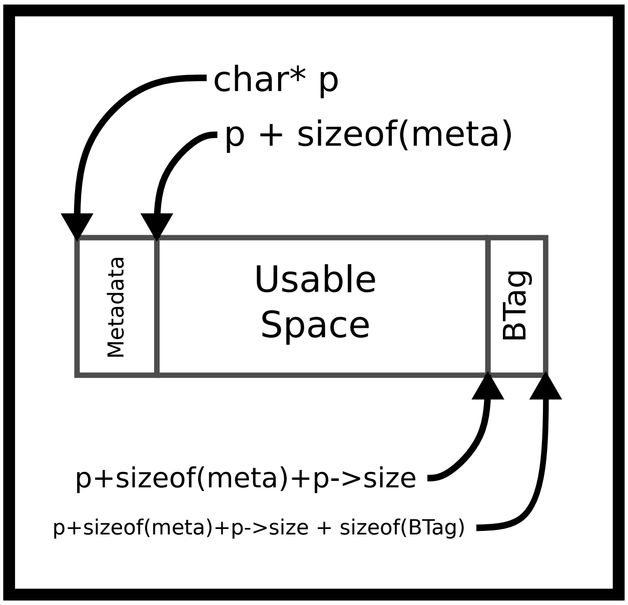


Figure 5.6: Malloc addition

One can grab the next block by finding the end of the current one. That is what we mean by “implicit list”.

The actual spacing may be different. The metadata can contain different things. A minimal metadata implementation would simply have the size of the block.

Since we write integers and pointers into memory that we already control, we can later consistently hop from one address to the next. This internal information represents some overhead. Meaning even if we had requested 1024 KiB of contiguous memory from the system, we an allocation of that size will fail.

Our heap memory is a list of blocks where each block is either allocated or unallocated. Thus there is conceptually a list of free blocks, but it is implicit in the form of block size information that we store as part of each block. Let’s think of it in terms of a simple implementation.

```
typedef struct {
  size_t block_size;
  char data[0];
} block;
block *p = sbrk(100);
p->size = 100 - sizeof(*p) - sizeof(BTag);
// Other block allocations
```

We could navigate from one block to the next block by adding the block’s size.

```
p + sizeof(metadata) + p->block_size + sizeof(BTag)
```

Make sure to get your casting right! Otherwise, the program will move an extreme amount of bytes over.

The calling program never sees these values. They are internal to the implementation of the memory allocator. As an example, suppose your allocator is asked to reserve 80 bytes (`malloc(80)`) and requires 8 bytes of internal header data. The allocator would need to find an unallocated space of at least 88 bytes. After updating the heap data it would return a pointer to the block. However, the returned pointer points to the usable space, not the internal data! Instead, we would return the start of the block + 8 bytes. In the implementation, remember that pointer arithmetic depends on type. For example, `p += 8` adds `8 * sizeof(p)`, not necessarily 8 bytes!

Implementing a Memory Allocator

The simplest implementation uses first fit. Start at the first block, assuming it exists, and iterate until a block that represents unallocated space of sufficient size is found, or we've checked all the blocks. If no suitable block is found, it's time to call `sbrk()` again to sufficiently extend the size of the heap. For the purposes of this class, we will try to serve every memory request until the operating system tells us we are going to run out of heap space. Other applications may limit themselves to a certain heap size and cause requests to intermittently fail. In addition, a fast implementation might extend it a significant amount so that we will not need to request more heap memory soon.

When a free block is found, it may be larger than the space we need. If so, we will create two entries in our implicit list. The first entry is the allocated block, the second entry is the remaining space. There are ways to do this if the program wants to keep the overhead small. We recommend first for going with readability.

```
typedef struct {
  size_t block_size;
  int is_free;
  char data[0];
} block;
block *p = sbrk(100);
p->size = 100 - sizeof(*p) - sizeof(boundary_tag);
// Other block allocations
```

If the program wants certain bits to hold different pieces of information, use bit fields!

```
typedef struct {
  unsigned int block_size : 7;
  unsigned int is_free : 1;
} size_free;

typedef struct {
  size_free info;
  char data[0];
} block;
```

The compiler will handle the shifting. After setting up your fields then it becomes simply looping through each of the blocks and checking the appropriate fields

Here is a visual representation of what happens. If we assume that we have a block that looks like this, we want to spit if the allocation is let's say 16 bytes The split we'll have to do is the following.

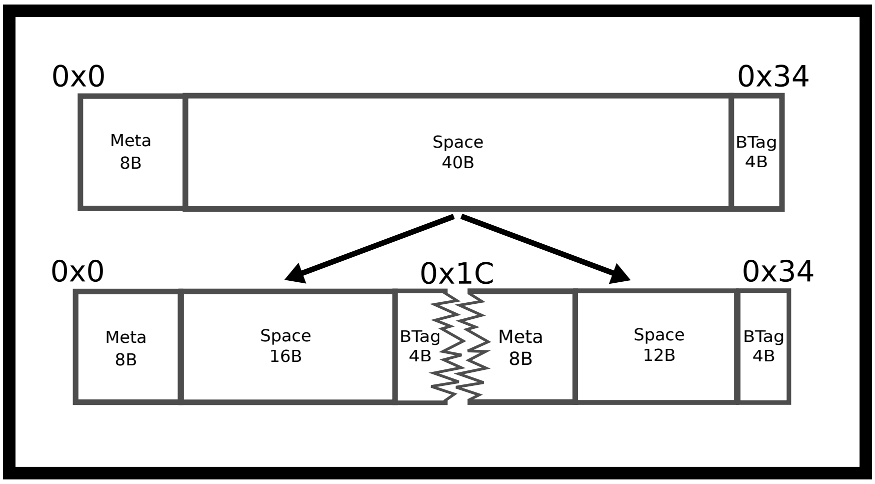


Figure 5.7: Malloc split

This is before alignment concerns as well.

Alignment and rounding up considerations

Many architectures expect multibyte primitives to be aligned to some multiple of 2 (4, 16, etc). For example, it's common to require 4-byte types to be aligned to 4-byte boundaries and 8-byte types on 8-byte boundaries. If multi-byte primitives are stored on an unreasonable boundary, the performance can be significantly impacted because it may require an additional memory read. On some architectures the penalty is even greater - the program will crash with a bus error. Most of you have experienced this in your architecture classes if there was no memory protection.

As malloc does not know how the user will use the allocated memory, the pointer returned to the program needs to be aligned for the worst case, which is architecture dependent.

From glibc documentation, the glibc malloc uses the following heuristic [1]

> The block that malloc gives you is guaranteed to be aligned so that it can hold any type of data. On GNU systems, the address is always a multiple of eight on most systems and a multiple of 16 on 64-bit systems." For example, if you need to calculate how many 16 byte units are required, don't forget to round up.

This is what the math would look like in C.

```
int s = (requested_bytes + tag_overhead_bytes + 15) / 16
```

The additional constant ensures incomplete units are rounded up. Note, real code is more likely to symbol sizes e.g. sizeof(x) - 1, rather than coding numerical constant 15. Here's a great article on memory alignment, if you are further interested

Another added effect could be internal fragmentation happens when the given block is larger than their allocation size. Let's say that we have a free block of size 16B (not including metadata). If they allocate 7 bytes, the allocator may want to round up to 16B and return the entire block. This gets sinister when implementing coalescing and splitting. If the allocator doesn't implement either, it may end up returning a block of size 64B for a 7B allocation! There is a *lot* of overhead for that allocation which is what we are trying to avoid.

Implementing free

When `free` is called we need to re-apply the offset to get back to the 'real' start of the block – to where we stored the size information. A naive implementation would simply mark the block as unused. If we are storing the block allocation status in a bitfield, then we need to clear the bit:

```
p->info.is_free = 0;
```

However, we have a bit more work to do. If the current block and the next block (if it exists) are both free we need to coalesce these blocks into a single block. Similarly, we also need to check the previous block, too. If that exists and represents an unallocated memory, then we need to coalesce the blocks into a single large block.

To be able to coalesce a free block with a previous free block we will also need to find the previous block, so we store the block's size at the end of the block, too. These are called "boundary tags" [5]. These are Knuth's solution to the coalescing problem both ways. As the blocks are contiguous, the end of one block sits right next to the start of the next block. So the current block (apart from the first one) can look a few bytes further back to look up the size of the previous block. With this information, the allocator can now jump backward!

Take for example a double coalesce. If we wanted to free the middle block we need to turn the surrounding blocks into one big blocks

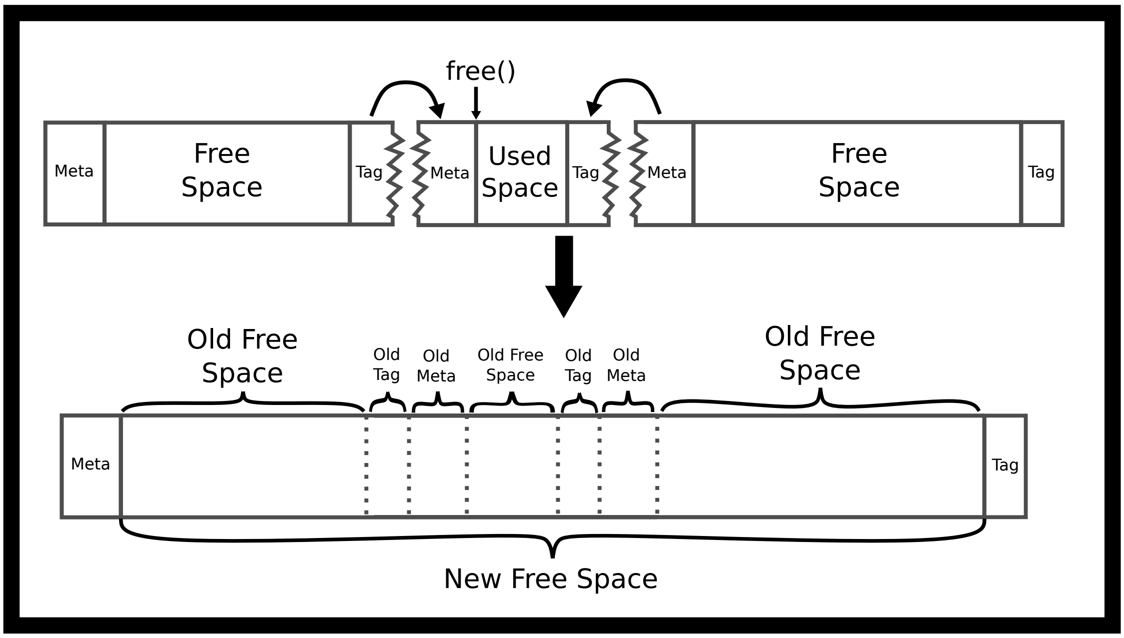


Figure 5.8: Free double coalesce

Performance

With the above description, it's possible to build a memory allocator. Its main advantage is simplicity - at least simple compared to other allocators! Allocating memory is a worst-case linear time operation – search linked lists for a sufficiently large free block. De-allocation is constant time. No more than 3 blocks will need to coalesce into a single block, and using a most recently used block scheme only one linked list entry.

Using this allocator it is possible to experiment with different placement strategies. For example, the allocator could start searching from the last deallocated a block. If the allocator stores pointers to blocks, it needs to update the pointers so that they always remain valid.

Explicit Free Lists Allocators

Better performance can be achieved by implementing an explicit doubly-linked list of free nodes. In that case, we can immediately traverse to the next free block and the previous free block. This can reduce the search time because the linked list only includes unallocated blocks. A second advantage is that we now have some control over the ordering of the linked list. For example, when a block is deallocated, we could choose to insert it into the beginning of the linked list rather than always between its neighbors. We may update our struct to look like this

```
typedef struct {
  size_t info;
  struct block *next;
  char data[0];
} block;
```

Here is what that would look like along with our implicit linked list

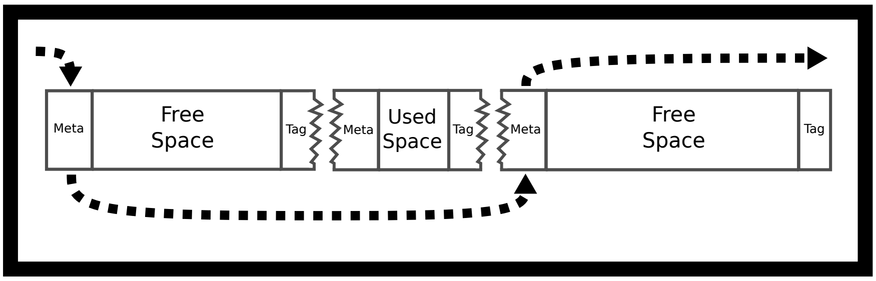


Figure 5.9: Free list

Where do we store the pointers of our linked list? A simple trick is to realize that the block itself is not being used and store the next and previous pointers as part of the block, though you have to ensure that the free blocks are always sufficiently large to hold two pointers. We still need to implement Boundary Tags, so we can correctly free blocks and coalesce them with their two neighbors. Consequently, explicit free lists require more code and complexity. With explicitly linked lists a fast and simple 'Find-First' algorithm is used to find the first sufficiently large link. However, since the link order can be modified, this corresponds to different placement strategies. If the links are maintained from largest to smallest, then this produces a 'Worst-Fit' placement strategy.

There are edge cases though, consider how to maintain your free list if also double coalescing. We've included a figure with a common mistake.

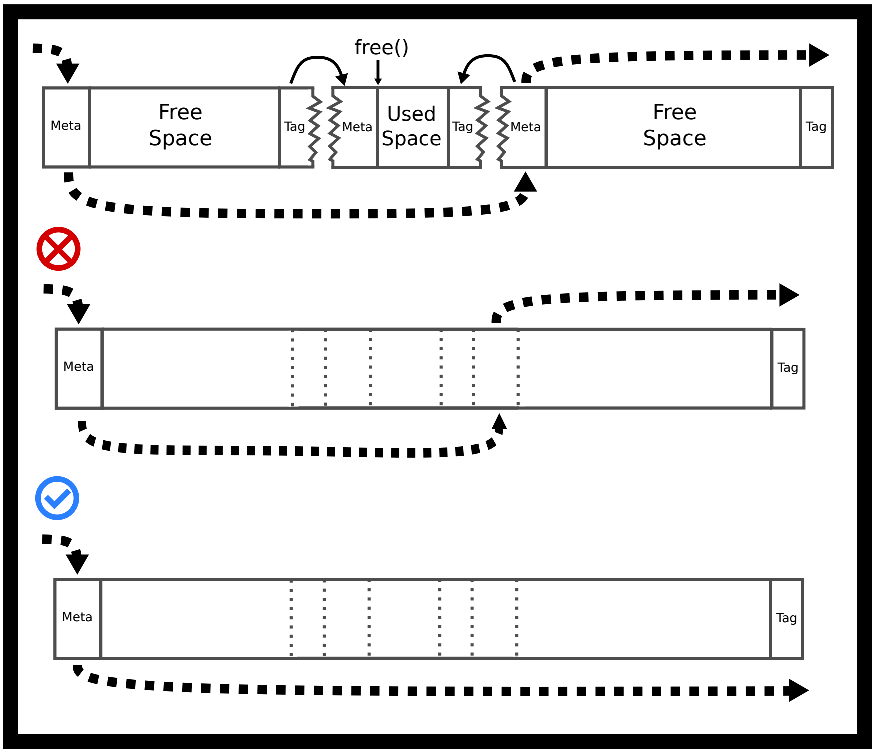


Figure 5.10: Free list good and bad coalesce

We recommend when trying to implement malloc that you draw out all the cases conceptually and then write the code.

Explicit linked list insertion policy

The newly deallocated block can be inserted easily into two possible positions: at the beginning or in address order. Inserting at the beginning creates a LIFO (last-in, first-out) policy. The most recently deallocated spaces will be reused. Studies suggest fragmentation is worse than using address order [7].

Inserting in address order ("Address ordered policy") inserts deallocated blocks so that the blocks are visited in increasing address order. This policy required more time to free a block because the boundary tags (size data) must be used to find the next and previous unallocated blocks. However, there is less fragmentation.

Case Study: Buddy Allocator, an example of a segregated list

A segregated allocator is one that divides the heap into different areas that are handled by different sub-allocators dependent on the size of the allocation request. Sizes are grouped into powers of two and each size is handled by a different sub-allocator and each size maintains its own free list.

A well-known allocator of this type is the buddy allocator [6, P. 85]. We'll discuss the binary buddy allocator which splits allocation into blocks of size $2^n; n = 1, 2, 3, ...$ times some base unit number of bytes, but others also exist like Fibonacci split where the allocation is rounded up to the next Fibonacci number. The basic concept is simple: If there are no free blocks of size 2^n, go to the next level and steal that block and split it into two. If two neighboring blocks of the same size become unallocated, they can coalesce together into a single large block of twice the size.

Buddy allocators are fast because the neighboring blocks to coalesce with can be calculated from the deallocated block's address, rather than traversing the size tags. Ultimate performance often requires a small amount of assembler code to use a specialized CPU instruction to find the lowest non-zero bit.

The main disadvantage of the Buddy allocator is that they suffer from *internal fragmentation* because allocations are rounded up to the nearest block size. For example, a 68-byte allocation will require a 128-byte block.

Case Study: SLUB Allocator, Slab allocation

The SLUB allocator is a slab allocator that serves different needs for the Linux kernel SLUB. Imagine you are creating an allocator for the kernel, what are your requirements? Here is a hypothetical shortlist.

1. First and foremost is you want a low memory footprint to have the kernel be able to be installed on all types of hardware: embedded, desktop, supercomputer, etc.

2. Then, you want the actual memory to be as contiguous as possible to make use of caching. Every time a system call is performed, the kernel's pages need to get loaded into memory. This means that if they are all contiguous, the processor will be able to cache them more efficiently

3. Lastly, you want your allocations to be fast.

Enter the SLUB allocator `kmalloc`. The SLUB allocator is a segregated list allocator with minimal splitting and coalescing. The difference here is that the segregated list focuses on more realistic allocation sizes, instead of powers of two. SLUB also focuses on low overall memory footprint while keeping pages in cache. There are blocks of different sizes and the kernel rounds up each allocation request to the lowest block size that satisfies it. One of the big differences between this allocator and the others is that it usually conforms to page sizes. We'll talk about virtual memory and pages in another chapter, but the kernel will be working with direct memory pages in spans of 4Kib or 4096 Bytes.

Further Reading

Guiding questions

- Is malloc'ed memory initialized? How about calloc'ed or realloc'ed memory?

- Does realloc accept a number of elements or bytes?

- Why may the allocation functions error?

See the man page or the appendix of the book 17.3.1!

- Slab Allocation

- Buddy Memory Allocation

Topics

- Best Fit
- Worst Fit
- First Fit
- Buddy Allocator
- Internal Fragmentation
- External Fragmentation
- sbrk
- Natural Alignment
- Boundary Tag
- Coalescing
- Splitting
- Slab Allocation/Memory Pool

Questions/Exercises

- What is Internal Fragmentation? When does it become an issue?
- What is External Fragmentation? When does it become an issue?
- What is a Best Fit placement strategy? How is it with External Fragmentation? Time Complexity?
- What is a Worst Fit placement strategy? Is it any better with External Fragmentation? Time Complexity?
- What is the First Fit Placement strategy? It's a little bit better with Fragmentation, right? Expected Time Complexity?
- Let's say that we are using a buddy allocator with a new slab of 64kb. How does it go about allocating 1.5kb?
- When does the 5 line `sbrk` implementation of malloc have a use?
- What is natural alignment?
- What is Coalescing/Splitting? How do they increase/decrease fragmentation? When can you coalesce or split?
- How do boundary tags work? How can they be used to coalesce or split?

Bibliography

[1] Virtual memory allocation and paging, May 2001. URL https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_3.html.

[2] Overview of malloc, Mar 2018. URL https://sourceware.org/glibc/wiki/MallocInternals.

[3] M. R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analysis of memory allocation algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 143–150, New York, NY, USA, 1972. ACM. doi: 10.1145/800152.804907. URL http://doi.acm.org/10.1145/800152.804907.

[4] Larry Jones. Wg14 n1539 committee draft iso/iec 9899: 201x, 2010.

[5] D.E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Number v. 1-2 in Addison-Wesley series in computer science and information processing. Addison-Wesley, 1973. ISBN 9780201038217. URL https://books.google.com/books?id=dC05RwAACAAJ.

[6] C.P. Rangan, V. Raman, and R. Ramanujam. *Foundations of Software Technology and Theoretical Computer Science: 19th Conference, Chennai, India, December 13-15, 1999 Proceedings*. FOUNDATIONS OF SOFTWARE TECHNOLOGY AND THEORETICAL COMPUTER SCIENCE. Springer, 1999. ISBN 9783540668367. URL https://books.google.com/books?id=0uHME7EfjQEC.

[7] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry G. Baler, editor, *Memory Management*, pages 1–116, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-45511-0.

6 Threads

> If you think your programs crashing before, wait until they crash ten times as fast
>
> Bhuvy

A thread is short for 'thread-of-execution'. It represents the sequence of instructions that the CPU has and will execute. To remember how to return from function calls, and to store the values of automatic variables and parameters a thread uses a stack. Almost weirdly, a thread is a process, meaning that creating a thread is similar to `fork`, except there is **no copying** meaning no copy on write. What this allows is for a process to share the same address space, variables, heap, file descriptors and etc. The actual system call to create a thread is similar to `fork`. It's `clone`. We won't go into the specifics, but you can read the man pages keeping in mind that it is outside the direct scope of this course. LWP or Lightweight Processes or threads are preferred to forking for a lot of scenarios because there is a lot less overhead creating them. But in some cases, notably python uses this, multiprocessing is the way to make your code faster.

Processes vs threads

Creating separate processes is useful when

- When more security is desired. For example, Chrome browser uses different processes for different tabs.
- When running an existing and complete program then a new process is required, for example starting 'gcc'.
- When you are running into synchronization primitives and each process is operating on something in the system.
- When you have too many threads – the kernel tries to schedule all the threads near each other which could cause more harm than good.
- When you don't want to worry about race conditions
- If one thread blocks in a task (say IO) then all threads block. Processes don't have that same restriction.
- When the amount of communication is minimal enough that simple IPC needs to be used.

On the other hand, creating threads is more useful when

- You want to leverage the power of a multi-core system to do one task
- When you can't deal with the overhead of processes
- When you want communication between the processes simplified
- When you want to threads to be part of the same process

Thread Internals

Your main function and other functions has automatic variables. We will store them in memory using a stack and keep track of how large the stack is by using a simple pointer (the "stack pointer"). If the thread calls another function, we move our stack pointer down, so that we have more space for parameters and automatic variables. Once it returns from a function, we can move the stack pointer back up to its previous value. We keep a copy of the old stack pointer value - on the stack! This is why returning from a function is quick. It's easy to 'free' the memory used by automatic variables because the program needs to change the stack pointer.

In a multi-threaded program, there are multiple stacks but only one address space. The pthread library allocates some stack space and uses the `clone` function call to start the thread at that stack address.

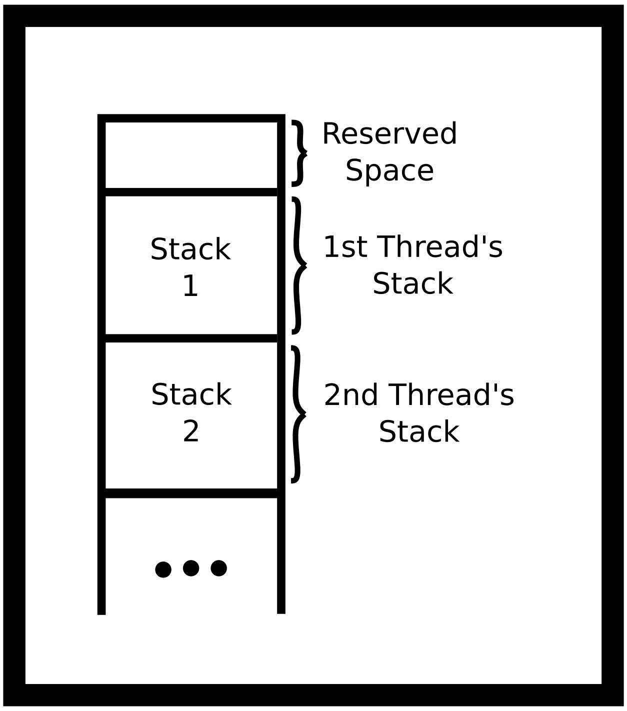


Figure 6.1: Thread stack visualization

A program can have more than one thread running inside a process. The programget the first thread for free! It runs the code you write inside 'main'. If a program need more threads, it can call `pthread_create` to create a new thread using the pthread library. You'll need to pass a pointer to a function so that the thread knows where to start.

The threads all live inside the same virtual memory because they are part of the same process. Thus they can all see the heap, the global variables, and the program code.

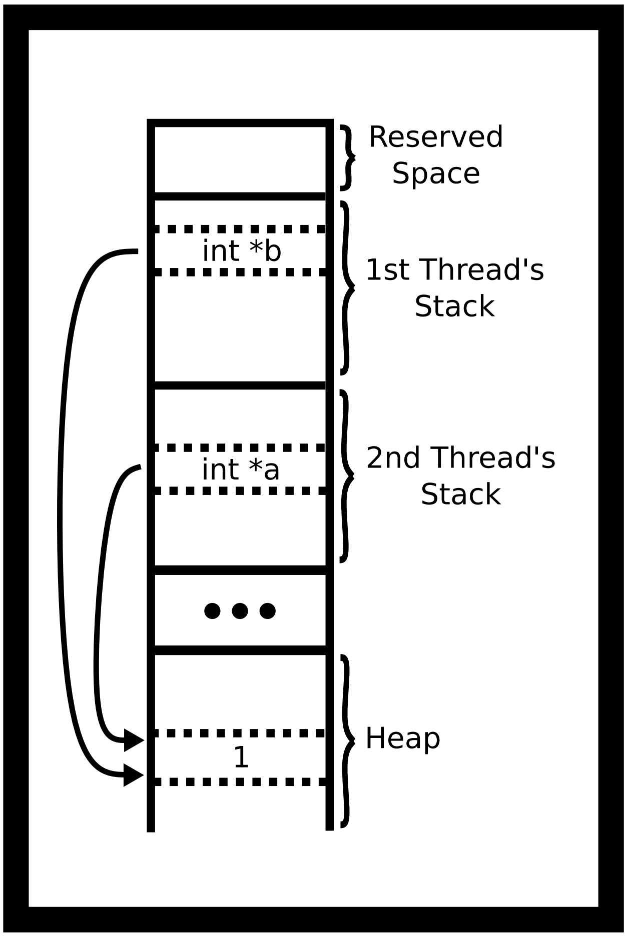


Figure 6.2: Threads pointing to the same place in the heap

Thus, a program can have two (or more) CPUs working on your program at the same time and inside the same process. It's up to the operating system to assign the threads to CPUs. If a program has more active threads than CPUs, the kernel will assign the thread to a CPU for a short duration or until it runs out of things to do and then will automatically switch the CPU to work on another thread. For example, one CPU might be processing the game AI while another thread is computing the graphics output.

Simple Usage

To use pthreads, include pthread.h and compile and link with -pthread or -lpthread compiler option. This option tells the compiler that your program requires threading support. To create a thread, use the function pthread_create. This function takes four arguments:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

- The first is a pointer to a variable that will hold the id of the newly created thread.
- The second is a pointer to attributes that we can use to tweak and tune some of the advanced features of pthreads.
- The third is a pointer to a function that we want to run
- Fourth is a pointer that will be given to our function

The argument `void *(*start_routine) (void *)` is difficult to read! It means a pointer that takes a `void *` pointer and returns a `void *` pointer. It looks like a function declaration except that the name of the function is wrapped with `(* .... )`

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

void *busy(void *ptr) {
  // ptr will point to "Hi"
  puts("Hello World");
  return NULL;
}
int main() {
  pthread_t id;
  pthread_create(&id, NULL, busy, "Hi");
  void *result;
  pthread_join(id, &result);
}
```

In the above example, the result will be `NULL` because the busy function returned `NULL`. We need to pass the address-of result because `pthread_join` will be writing into the contents of our pointer.

In the man pages, it warns that programmers should use `pthread_t` as an opaque type and not look at the internals. We do ignore that often, though.

Pthread Functions

Here are some common pthread functions.

- `pthread_create`. Creates a new thread. Every thread gets a new stack. If a program calls `pthread_create` twice, Your process will contain three stacks - one for each thread. The first thread is created when the process start, the other two after the create. Actually, there can be more stacks than this, but let's keep it simple. The important idea is that each thread requires a stack because the stack contains automatic variables and the old CPU PC register, so that it can go back to executing the calling function after the function is finished.
- `pthread_cancel` stops a thread. Note the thread may still continue. For example, it can be terminated

when the thread makes an operating system call (e.g. `write`). In practice, `pthread_cancel` is rarely used because a thread won't clean up open resources like files. An alternative implementation is to use a boolean (int) variable whose value is used to inform other threads that they should finish and clean up.

- `pthread_exit(void *)` stops the calling thread meaning the thread never returns after calling `pthread_exit`. The pthread library will automatically finish the process if no other threads are running. `pthread_exit(...)` is equivalent to returning from the thread's function; both finish the thread and also set the return value (void *pointer) for the thread. Calling `pthread_exit` in the `main` thread is a common way for simple programs to ensure that all threads finish. For example, in the following program, the `myfunc` threads will probably not have time to get started. On the other hand `exit()` exits the entire process and sets the process' exit value. This is equivalent to `return ();` in the main method. All threads inside the process are stopped. Note the `pthread_exit` version creates thread zombies; however, this is not a long-running process, so we don't care.

```
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, myfunc, "Jabberwocky");
  pthread_create(&tid2, NULL, myfunc, "Vorpel");
  if (keep_threads_going) {
    pthread_exit(NULL);
  } else {
    exit(42); //or return 42;
  }

  // No code is run after exit
}
```

- `pthread_join()` waits for a thread to finish and records its return value. Finished threads will continue to consume resources. Eventually, if enough threads are created, `pthread_create` will fail. In practice, this is only an issue for long-running processes but is not an issue for simple, short-lived processes as all thread resources are automatically freed when the process exits. This is equivalent to turning your children into zombies, so keep this in mind for long-running processes. In the exit example, we could also wait on all the threads.

```
// ...
void* result;
pthread_join(tid1, &result);
pthread_join(tid2, &result);
return 42;
// ...
```

There are many ways to exit threads. Here is a non-complete list.

- Returning from the thread function

- Calling pthread_exit
- Canceling the thread with pthread_cancel
- Terminating the process through a signal.
- calling exit() or abort()
- Returning from main
- Executing another program
- Unplugging your computer
- Some undefined behavior can terminate your threads, it is undefined behavior

Race Conditions

Race conditions are whenever the outcome of a program is determined by its sequence of events determined by the processor. This means that the execution of the code is non-deterministic. Meaning that the same program can run multiple times and depending on how the kernel schedules the threads could produce inaccurate results. The following is the canonical race condition.

```
void *thread_main(void *p) {
  int x = *p;
  x += x;
  *p = x;
  return NULL;
}

int main() {
  int data = 1;
  pthread_t one, two;
  pthread_create(&one, NULL, thread_main, &data);
  pthread_create(&two, NULL, thread_main, &data);
  pthread_join(one, NULL);
  pthread_join(two, NULL);
  printf("%d\n", data);
  return 0;
}
```

Breaking down the assembly there are many different accesses of the code. We will assume that data is stored in the eax register. The code to increment is the following with no optimization (assume int_ptr contains eax).

```
mov eax, DWORD PTR [rbp-4] ;Loads int_ptr
add eax, eax               ;Does the addition
```

```
mov DWORD PTR [rbp-4], eax ;Stores it back
```

Consider this access pattern.

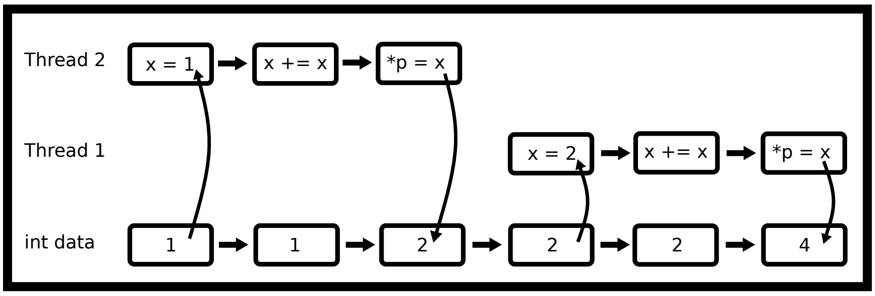


Figure 6.3: Thread access - not a race condition

This access pattern will cause the variable data to be 4. The problem is when the instructions are executed in parallel.

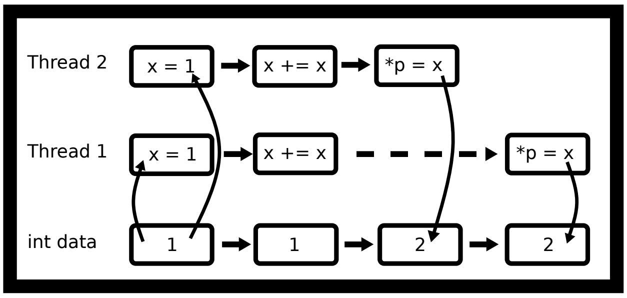


Figure 6.4: Thread access - race condition

This access pattern will cause the variable data to be 2. This is undefined behavior and a race condition. What we want is one thread to access the part of the code at a time.

But when compiled with -O2, assembly output is a single instruction.

```
shl dword ptr [rdi] # Optimized way of doing the add
```

Shouldn't that fix it? It is a single assembly instruction so no interleaving? It doesn't fix the problems that the *hardware itself* may experience a race condition because we as programmers didn't tell the hardware to check for it. The easiest way is to add the *lock* prefix [1, p. 1120].

But we don't want to be coding in assembly! We need to come up with a software solution to this problem.

A day at the races

Here is another small race condition. The following code is supposed to start ten threads with the integers 0 through 9 inclusive. However, when run prints out 1 7 8 8 8 8 8 8 8 10! Or seldom does it print out what we expect. Can you see why?

```
#include <pthread.h>
void* myfunc(void* ptr) {
  int i = *((int *) ptr);
  printf("%d ", i);
  return NULL;
}

int main() {
  // Each thread gets a different value of i to process
  int i;
  pthread_t tid;
  for(i =0; i < 10; i++) {
    pthread_create(&tid, NULL, myfunc, &i); // ERROR
  }
  pthread_exit(NULL);
}
```

The above code suffers from a race condition - the value of i is changing. The new threads start later in the example output the last thread starts after the loop has finished. To overcome this race-condition, we will give each thread a pointer to its own data area. For example, for each thread we may want to store the id, a starting value and an output value. We will instead treat i as a pointer and cast it by value.

```
void* myfunc(void* ptr) {
  int data = ((int) ptr);
  printf("%d ", data);
  return NULL;
}

int main() {
  // Each thread gets a different value of i to process
  int i;
  pthread_t tid;
  for(i =0; i < 10; i++) {
    pthread_create(&tid, NULL, myfunc, (void *)i);
  }
  pthread_exit(NULL);
}
```

Race conditions aren't in our code. They can be in provided code Some functions like asctime, getenv, strtok, strerror not thread-safe. Let's look at a simple function that is also not 'thread-safe'. The result buffer could be stored in global memory. This is good in a single-threaded program. We wouldn't want to return a pointer to an invalid address on the stack, but there's only one result buffer in the entire memory. If two threads were to use it at the same time, one would corrupt the other.

```
char *to_message(int num) {
  static char result [256];
  if (num < 10) sprintf(result, "%d : blah blah" , num);
  else strcpy(result, "Unknown");
  return result;
}
```

There are ways around this like using synchronization locks, but first let's do this by design. How would you fix the function above? You can change any of the parameters and any return types. Here is one valid solution.

```
int to_message_r(int num, char *buf, size_t nbytes) {
  size_t written;
  if (num < 10) {
    written = snprintf(buf, nbtytes, "%d : blah blah" , num);
  } else {
    strncpy(buf, "Unknown", nbytes);
    buf[nbytes] = '\0';
    written = strlen(buf) + 1;
  }
  return written <= nbytes;
}
```

Instead of making the function responsible for the memory, we made the caller responsible! A lot of programs, and hopefully your programs, have minimal communication needed. Often a malloc call is less work than locking a mutex or sending a message to another thread.

Don't Cross the Streams

A program can fork inside a process with multiple threads! However, the child process only has a single thread, which is a clone of the thread that called fork. We can see this as a simple example, where the background threads never print out a second message in the child process.

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

static pid_t child = -2;

void *sleepnprint(void *arg) {
  printf("%d:%s starting up...\n", getpid(), (char *) arg);
  while (child == -2) {sleep(1);} /* Later we will use condition
      variables */

  printf("%d:%s finishing...\n",getpid(), (char*)arg);

  return NULL;
}
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1,NULL, sleepnprint, "New Thread One");
  pthread_create(&tid2,NULL, sleepnprint, "New Thread Two");

  child = fork();
  printf("%d:%s\n",getpid(), "fork()ing complete");
  sleep(3);

  printf("%d:%s\n",getpid(), "Main thread finished");

  pthread_exit(NULL);
  return 0; /* Never executes */
}
```

```
8970:New Thread One starting up...
8970:fork()ing complete
8973:fork()ing complete
8970:New Thread Two starting up...
8970:New Thread Two finishing...
8970:New Thread One finishing...
8970:Main thread finished
8973:Main thread finished
```

In practice, creating threads before forking can lead to unexpected errors because (as demonstrated above) the other threads are immediately terminated when forking. Another thread might have locked a mutex like by calling malloc and never unlock it again. Advanced users may find `pthread_atfork` useful however we suggest a program avoid creating threads before forking unless you fully understand the limitations and difficulties of this approach.

Embarrassingly Parallel Problems

The study of parallel algorithms has exploded over the past few years. An embarrassingly parallel problem is any problem that needs little effort to turn parallel. A lot of them have some synchronization concepts with them but not always. You already know a parallelizable algorithm, Merge Sort!

```
void merge_sort(int *arr, size_t len){
  if(len > 1){
    // Merge Sort the left half
    // Merge Sort the right half
    // Merge the two halves
  }
```

With your new understanding of threads, all you need to do is create a thread for the left half, and one for the right half. Given that your CPU has multiple real cores, you will see a speedup following Amdahl's Law. The time complexity analysis gets interesting here as well. The parallel algorithm runs in $O(\log^3(n))$ running time because we have the analysis assumes that we have a lot of cores.

In practice though, we typically do two changes. One, once the array gets small enough, we ditch the Parallel Merge Sort algorithm and do conventional sort that works fast on small arrays, usually cache coherency rules at this level. The other thing that we know is that CPUs don't have infinite cores. To get around that, we typically keep a worker pool. You won't see the speedup right away because of things like cache coherency and scheduling extra threads. Over the bigger pieces of code though, you will start to see speedups.

Another embarrassingly parallel problem is parallel map. Say we want to apply a function to an entire array, one element at a time.

```
int *map(int (*func)(int), int *arr, size_t len){
  int *ret = malloc(len*sizeof(*arr));
  for(size_t i = 0; i < len; ++i) {
    ret[i] = func(arr[i]);
  }
  return ret;
}
```

Since none of the elements depend on any other element, how would you go about parallelizing this? What do you think would be the best way to split up the work between threads.

Extra: Scheduling

There are a few ways to split up the work. These are common to the OpenMP framework [2].

- `static scheduling` breaks up the problems into fixed-size chunks (predetermined) and have each thread work on each of the chunks. This works well when each of the subproblems takes roughly the same time because there is no additional overhead. All you need to do is write a loop and give the map function to each sub-array.

- `dynamic scheduling` as a new problem becomes available to have a thread serve it. This is useful when you don't know how long the scheduling will take

- `guided scheduling` This is a mix of the above with a mix of the benefits and tradeoffs. You start with static scheduling and move slowly to dynamic if needed

- `runtime scheduling` You have absolutely no idea how long the problems are going to take. Instead of deciding it yourself, let the program decide what to do!

No need to memorize any of the scheduling routines though. Openmp is a standard that is an alternative to pthreads. For example, here is how to parallelize a for loop

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  // Do stuff
}

// Specify the scheduling as follows
// #pragma omp parallel for scheduling(static)
```

Static scheduling will divide the problem into fixed-size chunks Dynamic scheduling will give a job once the loop is over Guided scheduling is Dynamic with chunks Runtime is a whole bag of worms.

Other Problems

From Wikipedia

- Serving static files on a web server to multiple users at once.
- The Mandelbrot set, Perlin noise, and similar images, where each point is calculated independently.
- Rendering of computer graphics. In computer animation, each frame may be rendered independently (see parallel rendering).
- Brute-force searches in cryptography.
- Notable real-world examples include distributed.net and proof-of-work systems used in cryptocurrency.
- BLAST searches in bioinformatics for multiple queries (but not for individual large queries)
- Large scale facial recognition systems that compare thousands of arbitrary acquired faces (e.g., a security or surveillance video via closed-circuit television) with a similarly large number of previously stored faces (e.g., a rogues gallery or similar watch list).
- Computer simulations comparing many independent scenarios, such as climate models.
- Evolutionary computation meta-heuristics such as genetic algorithms.
- Ensemble calculations of numerical weather prediction.
- Event simulation and reconstruction in particle physics.
- The marching squares algorithm
- Sieving step of the quadratic sieve and the number field sieve.
- Tree growth step of the random forest machine learning technique.
- Discrete Fourier Transform where each harmonic is independently calculated.

Extra: threads.h

We have a lot of threading libraries discussed in the extra section. We have the standard POSIX threads, OpenMP threads, we also have a new C11 threading library that is built into the standard. This library provides restricted functionality.

Why use restricted functionality? The key is in the name. Since this is the C standard library, it has to be implemented in all operating systems that are compliant which are pretty much all of them. This means there is first-class portability when using threads.

We won't drone on about the functions. Most of them are renaming of pthread functions anyway. If you ask why we don't teach these, there are a few reasons

1. They are pretty new. Even though the standard came out in roughly 2011, POSIX threads have been around forever. A lot of their quirks have been ironed out.

2. You lose expressivity. This is a concept that we'll talk about in later chapters, but when you make something portable, you lose some expressivity with the host hardware. That means that the threads.h library is pretty bare bones. It is hard to set CPU affinities. Schedule threads together. Efficiently look at the internals for performance reasons.

3. A lot of legacy code is already written with POSIX threads in mind. Other libraries like OpenMP, CUDA, MPI will either use POSIX processes or POSIX threads with a begrudging port to Windows.

Extra: Lightweight Processes?

In the beginning of the chapter, we mentioned that threads are processes. What do we mean by that? You can create a thread like a process Take a look at the example code below

```
// 8 KiB stacks
#define STACK_SIZE (8 * 1024 * 1024)

int thread_start(void *arg) {
  // Just like the pthread function
  puts("Hello Clone!")
  // This share the same heap and address space!
  return 0;
}

int main() {
  // Allocate stack space for the child
  char *child_stack = malloc(STACK_SIZE);
  // Remember stacks work by growing down, so we need
  // to give the top of the stack
  char *stack_top = stack + STACK_SIZE;

  // clone create thread
  pid_t pid = clone(thread_start, stack_top, SIGCHLD, NULL);
  if (pid == -1) {
    perror("clone");
    exit(1);
  }
  printf("Child pid %ld\n", (long) pid);

  // Wait like any child
  if (waitpid(pid, NULL, 0) == -1) {
    perror("waitpid");
    exit(1);
  }

  return 0;
}
```

It seems pretty simple right? Why not use this functionality? First, there is a decent bit of boilerplate code. In addition, pthreads are part of the POSIX standard and have defined functionality. Pthreads let a program set various attributes – some that resemble the option in clone – to customize your thread. But as we mentioned earlier, with each later of abstraction for portability reasons we lose some functionality. clone can do some neat things like keeping different parts of your heap the same while creating copies of other pages. A program has finer control of scheduling because it is a process with the same mappings.

At no time in this course should you be using clone. But in the future, know that it is a perfectly viable alternative to fork. You have to be careful and research edge cases.

Further Reading

Guiding questions

- What is the first argument to pthread create?
- What is the start routing in pthread create? How about arg?
- Why might pthread create fail?
- What are a few things that threads share in a process? What are a few things that threads have different?
- How can a thread uniquely identify itself?
- What are some examples of non thread safe library functions? Why might they not be thread safe?
- How can a program stop a thread?
- How can a program get back a thread's "return value"?

- man page
- pthread reference guide
- Concise third party sample code explaining create, join and exit

Topics

- pthread life-cycle
- Each thread has a stack
- Capturing return values from a thread
- Using `pthread_join`
- Using `pthread_create`
- Using `pthread_exit`
- Under what conditions will a process exit

Questions

- What happens when a pthread gets created?
- Where is each thread's stack?
- How does a program get a return value given a `pthread_t`? What are the ways a thread can set that return value? What happens if a program discards the return value?
- Why is `pthread_join` important (think stack space, registers, return values)?
- What does `pthread_exit` do if it is not the last thread? What other functions are called when after calling pthread_exit?
- Give me three conditions under which a multi-threaded process will exit. Are there any more?
- What is an embarrassingly parallel problem?

Bibliography

[1] Part Guide. Intel® 64 and ia-32 architectures software developers manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[2] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 2005. ISBN 9780471694663. URL `https://books.google.com/books?id=FH8fAQAAIAAJ`.

7

Synchronization

When multithreading gets interesting

Bhuvy

Synchronization coordinates various tasks so that they all finishin the the correct state. In C, we have series of mechanisms to control what threads are allowed to perform at a given state. Most of the time, the threads can progress without having to communicate, but every so often two or more threads may want to access a critical section. A critical section is a section of code that can only be executed by one thread at a time if the program is to function correctly. If two threads (or processes) were to execute code inside the critical section at the same time, it is possible that the program may no longer have the correct behavior.

As we said in the previous chapter, race conditions happen when an operation touches a piece of memory at the same time as another thread. If the memory location is only accessible by one thread, for example the automatic variable i below, then there is no possibility of a race condition and no Critical Section associated with i. However, the sum variable is a global variable and accessed by two threads. It is possible that two threads may attempt to increment the variable at the same time.

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

int sum = 0; //shared

void *countgold(void *param) {
  int i; //local to each thread
  for (i = 0; i < 10000000; i++) {
    sum += 1;
  }
  return NULL;
}

int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, countgold, NULL);
  pthread_create(&tid2, NULL, countgold, NULL);
  //Wait for both threads to finish:
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  printf("ARRRRG sum is %d\n", sum);
  return 0;
}
```

A typical output of the above code is `ARGGGH sum is <some number less than expected>` because there is a race condition. The code allows two threads to read and write `sum` at the same time. For example, both threads copy the current value of sum into CPU that runs each thread (let's pick 123). Both threads increment one to their own copy. Both threads write back the value (124). If the threads had accessed the sum at different times then the count would have been 125. A few of the possible different orderings are below.

Permissible Pattern

Table 7.1: Good Thread Access Pattern

Thread 1	Thread 2
Load Addr, Add 1 (i=1 locally)	...
Store (i=1 globally)	...
...	Load Addr, Add 1 (i=2 locally)
...	Store (i=2 globally)

Partial Overlap

Table 7.2: Bad Thread Access Pattern

Thread 1	Thread 2
Load Addr, Add 1 (i=1 locally)	...
Store (i=1 globally)	Load Addr, Add 1 (i=1 locally)
...	Store (i=1 globally)

Full Overlap

Table 7.3: Horrible Thread Access Pattern

Thread 1	Thread 2
Load Addr, Add 1 (i=1 locally)	Load Addr, Add 1 (i=1 locally)
Store (i=1 globally)	Store (i=1 globally)

We would like the first pattern of the code being mutually exclusive. Which leads us to our first synchronization primitive, a Mutex.

Mutex

To ensure that only one thread at a time can access a global variable, use a mutex – short for Mutual Exclusion. If one thread is currently inside a critical section we would like another thread to wait until the first thread is complete. A mutex isn't a primitive in the truest sense, though it is one of the smallest that has useful threading API. A mutex also isn't a data structure. It is an abstract data type. There are many ways to implement a mutex, and we'll give a few in this chapter. For right now let's use the black box that the pthread library gives us. Here is how we declare a mutex.

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER; // global variable
pthread_mutex_lock(&m); // start of Critical Section
// Critical section
pthread_mutex_unlock(&m); //end of Critical Section
```

Mutex Lifetime

There are a few ways of initializing a mutex. A program can use the macro `PTHREAD_MUTEX_INITIALIZER` only for global ('static') variables. `m = PTHREAD_MUTEX_INITIALIZER` is functionally equivalent to the more general purpose `pthread_mutex_init(m,NULL)`. The init version includes options to trade performance for additional error-checking and advanced sharing options. The init version also makes sure that the mutex is correctly initialized after the call, global mutexes are initialized on the first lock. A program can also call the init function inside of a program for a mutex located on the heap.

```
pthread_mutex_t *lock = malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(lock, NULL);
//later
pthread_mutex_destroy(lock);
free(lock);
```

Once we are finished with the mutex we should also call `pthread_mutex_destroy(m)` too. Note, a program can only destroy an unlocked mutex, destroy on a locked mutex is undefined behavior. Things to keep in mind about `init` and `destroy` A program doesn't need to destroy a mutex created with the global initializer.

1. Multiple threads init/destroy has undefined behavior
2. Destroying a locked mutex has undefined behavior
3. Keep to the pattern of one and only one thread initializing a mutex.
4. Copying the bytes of the mutex to a new memory location and then using the copy is *not* supported. To reference a mutex, a program *must* to have a pointer to that memory address.

Mutex Usages

How does one use a mutex? Here is a complete example in the spirit of the earlier piece of code.

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

// Create a mutex this ready to be locked!
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int sum = 0;

void *countgold(void *param) {
  int i;

  //Same thread that locks the mutex must unlock it
  //Critical section is ’sum += 1’
  //However locking and unlocking a million times
  //has significant overhead

  pthread_mutex_lock(&m);

  // Other threads that call lock will have to wait until we call unlock

  for (i = 0; i < 10000000; i++) {
    sum += 1;
  }
  pthread_mutex_unlock(&m);
  return NULL;
}

int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, countgold, NULL);
  pthread_create(&tid2, NULL, countgold, NULL);

  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  printf("ARRRRG sum is %d\n", sum);
  return 0;
}
```

In the code above, the thread gets the lock to the counting house before entering. The critical section is only the sum+=1 so the following version is also correct.

```
for (i = 0; i < 10000000; i++) {
  pthread_mutex_lock(&m);
  sum += 1;
  pthread_mutex_unlock(&m);
}
return NULL;
}
```

This process runs slower because we lock and unlock the mutex a million times, which is expensive - at least compared with incrementing a variable. In this simple example, we didn't need threads - we could have added up twice! A faster multi-thread example would be to add one million using an automatic (local) variable and only then adding it to a shared total after the calculation loop has finished:

```
int local = 0;
for (i = 0; i < 10000000; i++) {
  local += 1;
}

pthread_mutex_lock(&m);
sum += local;
pthread_mutex_unlock(&m);

return NULL;
}
```

If you know the Gaussian sum, you can avoid race conditions altogether, but this is for illustration.

Starting with the gotchas. Firstly, C Mutexes do not lock variables. A mutex is a simple data structure. It works with code, not data. If a mutex is locked, the other threads will continue. It's only when a thread attempts to lock a mutex that is already locked, will the thread have to wait. As soon as the original thread unlocks the mutex, the second (waiting) thread will acquire the lock and be able to continue. The following code creates a mutex that does effectively nothing.

```
int a;
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER,
m2 = = PTHREAD_MUTEX_INITIALIZER;
// later
// Thread 1
pthread_mutex_lock(&m1);
a++;
pthread_mutex_unlock(&m1);

// Thread 2
pthread_mutex_lock(&m2);
```

```
a++;
pthread_mutex_unlock(&m2);
```

Here are some other gotchas in no particular order

1. Don't cross the streams! If using threads, don't fork in the middle of your program. This means any time after your mutexes have been initialized.
2. The thread that locks a mutex is the only thread that can unlock it.
3. Each program can have multiple mutex locks. A thread safe design might include a lock with each data structure, one lock per heap, or one lock per set of data structures If a program has only one lock, then there may be significant contention for the lock. If two threads were updating two different counters, it isn't necessary to use the same lock.
4. Locks are only tools. They don't spot critical sections!
5. There will always be a small amount of overhead of calling `pthread_mutex_lock` and `pthread_mutex_unlock`. However, this is the price to pay for correctly functioning programs!
6. Not unlocking a mutex due to an early return during an error condition
7. Resource leak (not calling `pthread_mutex_destroy`)
8. Using an uninitialized mutex or using a mutex that has already been destroyed
9. Locking a mutex twice on a thread without unlocking first
10. Deadlock

Mutex Implementation

So we have this cool data structure. How do we implement it? A naive, incorrect implementation is shown below. The `unlock` function simply unlocks the mutex and returns. The lock function first checks to see if the lock is already locked. If it is currently locked, it will keep checking again until another thread has unlocked the mutex. For the time being, we'll avoid the condition that other threads are able to unlock a lock they don't own and focus on the mutual exclusion aspect.

```
// Version 1 (Incorrect!)

void lock(mutex_t *m) {
  while(m->locked) {/*Locked? Never-mind - loop and check again!*/ }

  m->locked = 1;
}

void unlock(mutex_t *m) {
  m->locked = 0;
}
```

Version 1 uses ‘busy-waiting’ unnecessarily wasting CPU resources. However, there is a more serious problem. We have a race-condition! If two threads both called lock concurrently, it is possible that both threads would read m_locked as zero. Thus both threads would believe they have exclusive access to the lock and both threads will continue.

We might attempt to reduce the CPU overhead a little by calling pthread_yield() inside the loop - pthread_yield suggests to the operating system that the thread does not use the CPU for a short while, so the CPU may be assigned to threads that are waiting to run. This still leaves the race-condition. We need a better implementation. We will talk about this later in the critical section part of this chapter. For now, we will talk about semaphores.

Extra: Implementing a Mutex with hardware

We can use C11 Atomics to do that perfectly! A complete solution is detailed here. This is a spinlock mutex, futex implementations can be found online.

First the data structure and initialization code.

```
typedef struct mutex_{
  // We need some variable to see if the lock is locked
  atomic_int_least8_t lock;
  // A mutex needs to keep track of its owner so
  // Another thread can’t unlock it
  pthread_t owner;
} mutex;

#define UNLOCKED 0
#define LOCKED 1
#define UNASSIGNED_OWNER 0

int mutex_init(mutex* mtx){
  // Some simple error checking
  if(!mtx){
    return 0;
  }
  // Not thread-safe the user has to take care of this
  atomic_init(&mtx->lock, UNLOCKED);
  mtx->owner = UNASSIGNED_OWNER;
  return 1;
}
```

This is the initialization code, nothing fancy here. We set the state of the mutex to unlocked and set the owner to locked.

```
int mutex_lock(mutex* mtx){
  int_least8_t zero = UNLOCKED;
  while(!atomic_compare_exchange_weak_explicit
  (&mtx->lock,
  &zero,
  LOCKED,
  memory_order_seq_cst,
  memory_order_seq_cst)){
    zero = UNLOCKED;
    sched_yield(); // Use system calls for scheduling speed
  }
  // We have the lock now
  mtx->owner = pthread_self();
  return 1;
}
```

What does this code do? It initializes a variable that we will keep as the unlocked state. Atomic Compare and Exchange is an instruction supported by most modern architectures (on x86 it's `lock cmpxchg`). The pseudocode for this operation looks like this

```
int atomic_compare_exchange_pseudo(int* addr1, int* addr2, int
    val){
  if(*addr1 == *addr2){
    *addr1 = val;
    return 1;
  }else{
    *addr2 = *addr1;
    return 0;
  }
}
```

Except it is all done *atomically* meaning in one uninterruptible operation. What does the *weak* part mean? Atomic instructions are prone to **spurious failures** meaning that there are two versions to these atomic functions a *strong* and a *weak* part, strong guarantees the success or failure while weak may fail even when the operation succeeds. These are the same spurious failures that you'll see in condition variables below. We are using weak because weak is faster, and we are in a loop! That means we are okay if it fails a little bit more often because we will keep spinning around anyway.

Inside the while loop, we have failed to grab the lock! We reset zero to unlocked and sleep for a little while. When we wake up we try to grab the lock again. Once we successfully swap, we are in the critical section! We set the mutex's owner to the current thread for the unlock method and return successfully.

How does this guarantee mutual exclusion? When working with atomics we are unsure! But in this simple example, we can because the thread that can successfully expect the lock to be UNLOCKED (0) and swap it to a LOCKED (1) state is considered the winner. How do we implement unlock?

```
int mutex_unlock(mutex* mtx){
  if(unlikely(pthread_self() != mtx->owner)){
    return 0; // Can't unlock a mutex if the thread isn't the owner
  }
  int_least8_t one = 1;
  //Critical section ends after this atomic
  mtx->owner = UNASSIGNED_OWNER;
  if(!atomic_compare_exchange_strong_explicit(
  &mtx->lock,
  &one,
  UNLOCKED,
  memory_order_seq_cst,
  memory_order_seq_cst)){
    //The mutex was never locked in the first place
    return 0;
  }
  return 1;
}
```

To satisfy the API, a thread can't unlock the mutex unless the thread is the one who owns it. Then we unassign the mutex owner, because critical section is over after the atomic. We want a strong exchange because we don't want to block. We expect the mutex to be locked, and we swap it to unlock. If the swap was successful, we unlocked the mutex. If the swap wasn't, that means that the mutex was UNLOCKED and we tried to switch it from UNLOCKED to UNLOCKED, preserving the behavior of unlock.

What is this memory order business? We were talking about memory fences earlier, here it is! We won't go into detail because it is outside the scope of this course but in the scope of this article. We need consistency to make sure no loads or stores are ordered before or after. A program need to create dependency chains for more efficient ordering.

Semaphore

A semaphore is another synchronization primitive. It is initialized to some value. Threads can either `sem_wait` or `sem_post` which lowers or increases the value. If the value reaches zero and a wait is called, the thread will be blocked until a post is called.

Using a semaphore is as easy as using a mutex. First, decide if on the initial value, for example the number of remaining spaces in an array. Unlike pthread mutex there are no shortcuts to creating a semaphore - use `sem_init`.

```
#include <semaphore.h>

sem_t s;
int main() {
  sem_init(&s, 0, 10); // returns -1 (=FAILED) on OS X
  sem_wait(&s); // Could do this 10 times without blocking
  sem_post(&s); // Announce that we've finished (and one more
      resource item is available; increment count)
  sem_destroy(&s); // release resources of the semaphore
}
```

When using a semaphore, wait and post can be called from different threads! Unlike a mutex, the increment and decrement can be from different threads.

This becomes especially useful if you want to use a semaphore to implement a mutex. A mutex is a semaphore that always `waits` before it `posts`. Some textbooks will refer to a mutex as a binary semaphore. You do have to be careful to never add more than one to a semaphore or otherwise your mutex abstraction breaks. That is usually why a mutex is used to implement a semaphore and vice versa.

- Initialize the semaphore with a count of one.
- Replace `pthread_mutex_lock` with `sem_wait`
- Replace `pthread_mutex_unlock` with `sem_post`

```
sem_t s;
sem_init(&s, 0, 1);

sem_wait(&s);
// Critical Section
sem_post(&s);
```

But be warned, it isn't the same! A mutex can handle what we call lock inversion well. Meaning the following code breaks with a traditional mutex, but produces a race condition with threads.

```
// Thread 1
sem_wait(&s);
// Critical Section
sem_post(&s);

// Thread 2
// Some threads want to see the world burn
sem_post(&s);

// Thread 3
sem_wait(&s);
// Not thread-safe!
sem_post(&s);
```

If we replace it with mutex lock, it won't work now.

```
// Thread 1
mutex_lock(&s);
```

```
// Critical Section
mutex_unlock(&s);

// Thread 2
// Foiled!
mutex_unlock(&s);

// Thread 3
mutex_lock(&s);
// Now it's thread-safe
mutex_unlock(&s);
```

Also, binary semaphores are different than mutexes because one thread can unlock a mutex from a different thread.

Signal Safety

Also, sem_post is one of a handful of functions that can be correctly used inside a signal handler pthread_mutex_unlock is not. We can release a waiting thread that can now make all of the calls that we disallowed to call inside the signal handler itself e.g. printf. Here is some code that utilizes this;

```
#include <stdio.h>
#include <pthread.h>
#include <signal.h>
#include <semaphore.h>
#include <unistd.h>

sem_t s;

void handler(int signal) {
  sem_post(&s); /* Release the Kraken! */
}

void *singsong(void *param) {
  sem_wait(&s);
  printf("Waiting until a signal releases...\n");
}

int main() {
  int ok = sem_init(&s, 0, 0 /* Initial value of zero*/);
  if (ok == -1) {
    perror("Could not create unnamed semaphore");
    return 1;
  }
  signal(SIGINT, handler); // Too simple! See Signals chapter
  pthread_t tid;
  pthread_create(&tid, NULL, singsong, NULL);
  pthread_exit(NULL); /* Process will exit when there are no more
      threads */
}
```

Other uses for semaphores are keeping track of empty spaces in arrays. We will discuss these in the thread-safe data structures section.

Condition Variables

Condition variables allow a set of threads to sleep until woken up. The API allows either one or all threads to be woken up. If a program only wakes one thread, the operating system will decide which thread to wake up. Threads don't wake threads other directly like by id. Instead, a thread 'signal's the condition variable, which then will wake up one (or all) threads that are sleeping inside the condition variable.

Condition variables are also used with a mutex and with a loop, so when woken up they have to check a condition in a critical section. If a thread needs to be woken up outside of a critical section, there are other ways to do this in POSIX. Threads sleeping inside a condition variable are woken up by calling `pthread_cond_broadcast` (wake up all) or `pthread_cond_signal` (wake up one). Note despite the function name, this has nothing to do with POSIX `signal`s!

Occasionally, a waiting thread may appear to wake up for no reason. This is called a *spurious wakeup*. If you read the hardware implementation of a mutex section, this is similar to the atomic failure of the same name.

Why do spurious wakeups happen? For performance. On multi-CPU systems, it is possible that a race condition could cause a wake-up (signal) request to be unnoticed. The kernel may not detect this lost wake-up call but can detect when it might occur. To avoid the potentially lost signal, the thread is woken up so that the program code can test the condition again.

Extra: Why do Condition Variables also need a mutex?

Condition variables need a mutex for a few reasons. One is simply that a mutex is needed to synchronize the changes of the *condition variable* across threads. Imagine a condition variable needing to provide its own internal synchronization to ensure its data structures work correctly. Often, we use a mutex to synchronize other parts of our code, so why double the cost of using a condition variable. Another example relates to high priority systems. Let's examine a code snippet.

```
// Thread 1
while (answer < 42) pthread_cond_wait(cv);

// Thread 2
answer = 42
pthread_cond_signal(cv);
```

Table 7.4: Signaling without Mutex

Thread 1	Thread 2
while(answer < 42)	
	answer++
	pthread_cond_signal(cv)
pthread_cond_wait(cv)	

The problem here is that a programmer expects the signal to wake up the waiting thread. Since instructions are allowed to be interleaved without a mutex, this causes an interleaving that is confusing to application designers. Note that technically the API of the condition variable is satisfied. The wait call *happens-after* the call to signal, and signal is only required to release at most a single thread whose call to wait *happened-before*.

Another problem is the need to satisfy real-time scheduling concerns which we only outline here. In a time-critical application, the waiting thread with the *highest priority* should be allowed to continue first. To satisfy this requirement the mutex must also be locked before calling `pthread_cond_signal` or `pthread_cond_broadcast`. For the curious, here is a longer, historical discussion.

Condition Wait Example

The call `pthread_cond_wait` performs three actions:

1. Unlock the mutex. The mutex must be locked.
2. Sleeps until `pthread_cond_signal` is called on the same condition variable.
3. Before returning, locks the mutex.

Condition variables are *always* used with a mutex lock. Before calling *wait*, the mutex lock must be locked and *wait* must be wrapped with a loop.

```
pthread_cond_t cv;
pthread_mutex_t m;
int count;

// Initialize
pthread_cond_init(&cv, NULL);
pthread_mutex_init(&m, NULL);
count = 0;

// Thread 1
pthread_mutex_lock(&m);
while (count < 10) {
  pthread_cond_wait(&cv, &m);
  /* Remember that cond_wait unlocks the mutex before blocking
      (waiting)! */
  /* After unlocking, other threads can claim the mutex. */
  /* When this thread is later woken it will */
  /* re-lock the mutex before returning */
}
pthread_mutex_unlock(&m);

//later clean up with pthread_cond_destroy(&cv); and mutex_destroy


// Thread 2:
while (1) {
  pthread_mutex_lock(&m);
  count++;
  pthread_cond_signal(&cv);
  /* Even though the other thread is woken up it cannot not return
      */
  /* from pthread_cond_wait until we have unlocked the mutex. This
      is */
  /* a good thing! In fact, it is usually the best practice to call
      */
  /* cond_signal or cond_broadcast before unlocking the mutex */
  pthread_mutex_unlock(&m);
}
```

This is a pretty naive example, but it shows that we can tell threads to wake up in a standardized manner. In the next section, we will use these to implement efficient blocking data structures.

Thread-Safe Data Structures

Naturally, we want our data structures to be thread-safe as well! We can use mutexes and synchronization primitives to make that happen. First a few definitions. Atomicity is when an operation is thread-safe. We have atomic instructions in hardware by providing the lock prefix

```
lock ...
```

But Atomicity also applies to higher orders of operations. We say a data structure operation is atomic if it happens all at once and successfully or not at all.

As such, we can use synchronization primitives to make our data structures thread-safe. For the most part, we will be using mutexes because they carry more semantic meaning than a binary semaphore. Note, this is an introduction. Writing high-performance thread-safe data structures requires its own book! Take for example the following thread-unsafe stack.

```
// A simple fixed-sized stack (version 1)
#define STACK_SIZE 20
```

```c
int count;
double values[STACK_SIZE];

void push(double v) {
  values[count++] = v;
}

double pop() {
  return values[--count];
}

int is_empty() {
  return count == 0;
}
```

Version 1 of the stack is thread-unsafe because if two threads call push or pop at the same time then the results or the stack can be inconsistent. For example, imagine if two threads call pop at the same time then both threads may read the same value, both may read the original count value.

To turn this into a thread-safe data structure we need to identify the *critical sections* of our code, meaning we need to ask which section(s) of the code must only have one thread at a time. In the above example the push, pop, and is_empty functions access the same memory and all critical sections for the stack. While push (and pop) is executing, the data structure is an inconsistent state, for example the count may not have been written to, so it may still contain the original value. By wrapping these methods with a mutex we can ensure that only one thread at a time can update (or read) the stack. A candidate 'solution' is shown below. Is it correct? If not, how will it fail?

```c
// An attempt at a thread-safe stack (version 2)
#define STACK_SIZE 20
int count;
double values[STACK_SIZE];

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;

void push(double v) {
  pthread_mutex_lock(&m1);
  values[count++] = v;
  pthread_mutex_unlock(&m1);
}

double pop() {
  pthread_mutex_lock(&m2);
  double v = values[--count];
  pthread_mutex_unlock(&m2);

  return v;
}

int is_empty() {
  pthread_mutex_lock(&m1);
  return count == 0;
  pthread_mutex_unlock(&m1);
}
```

Version 2 contains at least one error. Take a moment to see if you can the error(s) and work out the consequence(s).

If three threads called push() at the same time, the lock m1 ensures that only one thread at time manipulates the stack on push or is_empty – Two threads will need to wait until the first thread completes A similar argument applies to concurrent calls to pop. However, Version 2 does not prevent push and pop from running at the same time because push and pop use two different mutex locks. The fix is simple in this case - use the same mutex lock for both the push and pop functions.

The code has a second error. is_empty returns after the comparison and leaves the mutex unlocked. However, the error would not be spotted immediately. For example, suppose one thread calls is_empty and a second thread later calls push. This thread would mysteriously stop. Using debugger, you can discover that the thread is stuck at the lock() method inside the push method because the lock was never unlocked by the earlier is_empty call. Thus an oversight in one thread led to problems much later in time in an arbitrary other thread. Let's try to rectify these problems

```
// An attempt at a thread-safe stack (version 3)
int count;
double values[count];
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void push(double v) {
  pthread_mutex_lock(&m);
  values[count++] = v;
  pthread_mutex_unlock(&m);
}
double pop() {
  pthread_mutex_lock(&m);
  double v = values[--count];
  pthread_mutex_unlock(&m);
  return v;
}
int is_empty() {
  pthread_mutex_lock(&m);
  int result = count == 0;
  pthread_mutex_unlock(&m);
  return result;
}
```

Version 3 is thread-safe. We have ensured mutual exclusion for all of the critical sections. There are a few things to note.

- is_empty is thread-safe but its result may already be out-of-date. The stack may no longer be empty by the time the thread gets the result! This is usually why in thread-safe data structures, functions that return sizes are removed or deprecated.
- There is no protection against underflow (popping on an empty stack) or overflow (pushing onto an already-full stack)

The last point can be fixed using counting semaphores. The implementation assumes a single stack. A more general-purpose version might include the mutex as part of the memory structure and use pthread_mutex_init to initialize the mutex. For example,

```
// Support for multiple stacks (each one has a mutex)
typedef struct stack {
  int count;
  pthread_mutex_t m;
  double *values;
} stack_t;

stack_t* stack_create(int capacity) {
  stack_t *result = malloc(sizeof(stack_t));
  result->count = 0;
  result->values = malloc(sizeof(double) * capacity);
  pthread_mutex_init(&result->m, NULL);
  return result;
}
void stack_destroy(stack_t *s) {
  free(s->values);
  pthread_mutex_destroy(&s->m);
  free(s);
}

// Warning no underflow or overflow checks!

void push(stack_t *s, double v) {
  pthread_mutex_lock(&s->m);
  s->values[(s->count)++] = v;
  pthread_mutex_unlock(&s->m);
}

double pop(stack_t *s) {
  pthread_mutex_lock(&s->m);
  double v = s->values[--(s->count)];
  pthread_mutex_unlock(&s->m);
  return v;
}
```

```
int is_empty(stack_t *s) {
  pthread_mutex_lock(&s->m);
  int result = s->count == 0;
  pthread_mutex_unlock(&s->m);
  return result;
}

int main() {
  stack_t *s1 = stack_create(10 /* Max capacity*/);
  stack_t *s2 = stack_create(10);
  push(s1, 3.141);
  push(s2, pop(s1));
  stack_destroy(s2);
  stack_destroy(s1);
}
```

Before we fix the problems with semaphores. How would we fix the problems with condition variables? Try it out before you look at the code in the previous section. We need to wait in push and pop if our stack is full or empty respectively. Attempted solution:

```
// Assume cv is a condition variable
// correctly initialized

void push(stack_t *s, double v) {
  pthread_mutex_lock(&s->m);
  if(s->count == 0) pthread_cond_wait(&s->cv, &s->m);
  s->values[(s->count)++] = v;
  pthread_mutex_unlock(&s->m);
}

double pop(stack_t *s) {
  pthread_mutex_lock(&s->m);
  if(s->count == 0) pthread_cond_wait(&s->cv, &s->m);
  double v = s->values[--(s->count)];
  pthread_mutex_unlock(&s->m);
  return v;
}
```

Does the following solution work? Take a second before looking at the answer to spot the errors. So did you catch all of them?

1. The first one is a simple one. In push, our check should be against the total capacity, not zero.

2. We only have if statement checks. wait() could spuriously wake up

3. We never signal any of the threads! Threads could get stuck waiting indefinitely.

Let's fix those errors Does this solution work?

```
void push(stack_t *s, double v) {
  pthread_mutex_lock(&s->m);
  while(s->count == capacity) pthread_cond_wait(&s->cv, &s->m);
  s->values[(s->count)++] = v;
  pthread_mutex_unlock(&s->m);
  pthread_cond_signal(&s->cv);
}

double pop(stack_t *s) {
  pthread_mutex_lock(&s->m);
  while(s->count == 0) pthread_cond_wait(&s->cv, &s->m);
  double v = s->values[--(s->count)];
  pthread_cond_broadcast(&s->cv);
  pthread_mutex_unlock(&s->m);
  return v;
}
```

This solution doesn't work either! The problem is with the signal. Can you see why? What would you do to fix it?

Now, how would we use counting semaphores to prevent over and underflow? Let's discuss it in the next section.

Using Semaphores

Let's use a counting semaphore to keep track of how many spaces remain and another semaphore to track the number of items in the stack. We will call these two semaphores `sremain` and `sitems`. Remember `sem_wait` will wait if the semaphore's count has been decremented to zero (by another thread calling sem_post).

```
// Sketch #1

sem_t sitems;
sem_t sremain;
void stack_init(){
  sem_init(&sitems, 0, 0);
  sem_init(&sremain, 0, 10);
}


double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);
  ...
  void push(double v) {
    // Wait until there’s at least one space
    sem_wait(&sremain);
    ...
  }
```

Sketch #2 has implemented the post too early. Another thread waiting in push can erroneously attempt to write into a full stack. Similarly, a thread waiting in the pop() is allowed to continue too early.

```
// Sketch #2 (Error!)
double pop() {
  // Wait until there’s at least one item
  sem_wait(&sitems);
  sem_post(&sremain); // error! wakes up pushing() thread too early
  return values[--count];
}
void push(double v) {
  // Wait until there’s at least one space
  sem_wait(&sremain);
  sem_post(&sitems); // error! wakes up a popping() thread too early
  values[count++] = v;
}
```

Sketch 3 implements the correct semaphore logic, but can you spot the error?

```
// Sketch #3 (Error!)
double pop() {
  // Wait until there’s at least one item
  sem_wait(&sitems);
  double v= values[--count];
  sem_post(&sremain);
  return v;
}

void push(double v) {
  // Wait until there’s at least one space
  sem_wait(&sremain);
  values[count++] = v;
  sem_post(&sitems);
}
```

Sketch 3 correctly enforces buffer full and buffer empty conditions using semaphores. However, there is no *mutual exclusion*. Two threads can be in the *critical section* at the same time, which would corrupt the data structure or least lead to data loss. The fix is to wrap a mutex around the critical section:

```
// Simple single stack - see the above example on how to convert
    this into multiple stacks.
// Also a robust POSIX implementation would check for EINTR and
    error codes of sem_wait.

// PTHREAD_MUTEX_INITIALIZER for statics (use pthread_mutex_init()
    for stack/heap memory)
#define SPACES 10
pthread_mutex_t m= PTHREAD_MUTEX_INITIALIZER;
int count = 0;
double values[SPACES];
sem_t sitems, sremain;

void init() {
  sem_init(&sitems, 0, 0);
  sem_init(&sremains, 0, SPACES); // 10 spaces
}

double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);

  pthread_mutex_lock(&m); // CRITICAL SECTION
  double v= values[--count];
  pthread_mutex_unlock(&m);

  sem_post(&sremain); // Hey world, there's at least one space
  return v;
}

void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);

  pthread_mutex_lock(&m); // CRITICAL SECTION
  values[count++] = v;
  pthread_mutex_unlock(&m);

  sem_post(&sitems); // Hey world, there's at least one item
}
// Note a robust solution will need to check sem_wait's result for
    EINTR (more about this later)
```

What happens when we start inverting the lock and wait orders?

```
double pop() {
  pthread_mutex_lock(&m);
  sem_wait(&sitems);

  double v= values[--count];
  pthread_mutex_unlock(&m);

  sem_post(&sremain);
  return v;
}

void push(double v) {
  sem_wait(&sremain);

  pthread_mutex_lock(&m);
  values[count++] = v;
  pthread_mutex_unlock(&m);

  sem_post(&sitems);
}
```

Rather than giving you the answer, we'll let you think about this. Is this a permissible way to lock and unlock? Is there a series of operations that could cause a race condition? How about deadlock? If there is, provide it. If there isn't, provide a short justification proof of why that won't happen.

Software Solutions to the Critical Section

As already discussed, there are critical parts of our code that can only be executed by one thread at a time. We describe this requirement as 'mutual exclusion'. Only one thread (or process) may have access to the shared resource. In multi-threaded programs, we can wrap a critical section with mutex lock and unlock calls:

```
pthread_mutex_lock() // one thread allowed at a time! (others will
    have to wait here)
// ... Do Critical Section stuff here!
pthread_mutex_unlock() // let other waiting threads continue
```

How would we implement these lock and unlock calls? Can we create a pure software algorithm that assures mutual exclusion? Here is our attempt from earlier.

```
pthread_mutex_lock(p_mutex_t *m) {
  while(m->lock) ;
  m->lock = 1;
}
pthread_mutex_unlock(p_mutex_t *m) {
  m->lock = 0;
}
```

As we touched on earlier, this implementation *does not satisfy Mutual Exclusion* even considering that threads can unlock other threads locks. Let's take a close look at this 'implementation' from two threads running around the same time.

To simplify the discussion, we consider only two threads. Note these arguments work for threads and processes and the classic CS literature discusses these problems in terms of two processes that need exclusive access to a critical section or shared resource. Raising a flag represents a thread/process's intention to enter the critical section.

There are three main desirable properties that we desire in a solution to the critical section problem.

1. Mutual Exclusion. The thread/process gets exclusive access. Others must wait until it exits the critical section.

2. Bounded Wait. A thread/process cannot get superseded by another thread infinite amounts of time.

3. Progress. If no thread/process is inside the critical section, the thread/process should be able to proceed without having to wait.

With these ideas in mind, let's examine another candidate solution that uses a turn-based flag only if two threads both required access at the same time.

Naive Solutions

Remember that the pseudo-code outlined below is part of a larger program. The thread or process will typically need to enter the critical section many times during the lifetime of the process. So, imagine each example as wrapped inside a loop where for a random amount of time the thread or process is working on something else.

Is there anything wrong with the candidate solution described below?

```
// Candidate #1
wait until your flag is lowered
raise my flag
// Do Critical Section stuff
lower my flag
```

Answer: Candidate solution #1 also suffers from a race condition because both threads/processes could read each other's flag value as lowered and continue.

This suggests we should raise the flag *before* checking the other thread's flag, which is candidate solution #2 below.

```
// Candidate #2
raise my flag
wait until your flag is lowered
// Do Critical Section stuff
lower my flag
```

Candidate #2 satisfies mutual exclusion. It is impossible for two threads to be inside the critical section at the same time. However, this code suffers from deadlock! Suppose two threads wish to enter the critical section at the same time.

Table 7.5: Candidate Solution #2 Analysis

Time	Thread 1	Thread 2
1	Raise Flag	
2		Raise Flag
3	Wait	Wait

Both processes are now waiting for the other one to lower their flags. Neither one will enter the critical section as both are now stuck forever! This suggests we should use a turn-based variable to try to resolve who should proceed.

Turn-based solutions

The following candidate solution #3 uses a turn-based variable to politely allow one thread and then the other to continue

```
// Candidate #3
wait until my turn is myid
// Do Critical Section stuff
turn = yourid
```

Candidate #3 satisfies mutual exclusion. Each thread or process gets exclusive access to the Critical Section. However, both threads/processes must take a strict turn-based approach to use the critical section. They are forced into an alternating critical section access pattern. If thread 1 wishes to read a hash table every millisecond, but another thread writes to a hash table every second, then the reading thread would have to wait another 999ms before being able to read from the hash table again. This 'solution' is ineffective because our threads should be able to make progress and enter the critical section if no other thread is currently in the critical section.

Turn and Flag solutions

Is the following a correct solution to CSP?

```
\\ Candidate #4
raise my flag
if your flag is raised, wait until my turn
// Do Critical Section stuff
turn = yourid
lower my flag
```

Analyzing these solutions is tricky. Even peer-reviewed papers on this specific subject contain incorrect solutions [?]! At first glance, it appears to satisfy Mutual Exclusion, Bounded Wait and Progress The turn-based flag is only used in the event of a tie, so Progress and Bounded Wait is allowed and mutual exclusion appears to be satisfied. Perhaps you can find a counter-example?

Candidate #4 fails because a thread does not wait until the other thread lowers its flag. After some thought or inspiration, the following scenario can be created to demonstrate how Mutual Exclusion is not satisfied.

Imagine the first thread runs this code twice. The turn flag now points to the second thread. While the first thread is still inside the Critical Section, the second thread arrives. The second thread can immediately continue into the Critical Section!

Table 7.6: Candidate Solution #4

Time	Turn	Thread # 1	Thread # 2
1	2	Raise my flag	
2	2	If your flag is raised, wait until my turn	Raise my flag
3	2	// Do Critical Section Stuff	If your flag is raised, wait until my turn (TRUE!)
4	2	// Do Critical Section Stuff	Do Critical Section Stuff - OOPS

Working Solutions

The first solution to the problem was Dekker's Solution. Dekker's Algorithm (1962) was the first provably correct solution. Though, it was in an unpublished paper, so it was not discovered until later [1] (this is an English transcribed version released in 1965). A version of the algorithm is below.

```
raise my flag
while (your flag is raised) :
   if it is your turn to win :
     lower my flag
     wait while your turn
     raise my flag
// Do Critical Section stuff
set your turn to win
lower my flag
```

Notice how the process's flag is always raised during the critical section no matter if the loop is iterated zero, once or more times. Further, the flag can be interpreted as an immediate intent to enter the critical section. Only if the other process has also raised the flag will one process defer, lower their intent flag and wait. Let's check the conditions.

1. Mutual Exclusion. Let's try to sketch a simple proof. The loop invariant is that at the start of checking the condition, your flag has to be raised – this is by exhaustion. Since the only way that a thread can leave the loop is by having the condition be false, the flag must be raised for the entirety of the critical section. Since the loop prevents a thread from exiting while the other thread's flag is raised and a thread has its flag raised in the critical section, the other thread can't enter the critical section at the same time.
2. Bounded Wait. Assuming that the critical section ends in finite time, a thread once it has left the critical section cannot then get the critical section back. The reason being is the turn variable is set to the other thread, meaning that that thread now has priority. That means a thread cannot be superseded infinitely by another thread.
3. Progress. If the other thread isn't in the critical section, it will simply continue with a simple check. We didn't make any statement about if threads are randomly stopped by the system scheduler. This is an idealized scenario where threads will keep executing instructions.

Peterson's Solution

Peterson published his novel and surprisingly simple solution in 1981 [2]. A version of his algorithm is shown below that uses a shared variable turn.

```
// Candidate #5
raise my flag
turn = other_thread_id
while (your flag is up and turn is other_thread_id)
    loop
// Do Critical Section stuff
lower my flag
```

This solution satisfies Mutual Exclusion, Bounded Wait and Progress. If thread #2 has set turn to 2 and is currently inside the critical section. Thread #1 arrives, *sets the turn back to 1* and now waits until thread 2 lowers the flag.

1. Mutual Exclusion. Let's try to sketch a simple proof again. A thread doesn't get into the critical section until the turn variable is yours or the other thread's flag isn't up. If the other thread's flag isn't up, it isn't trying to enter the critical section. That is the first action the thread does and the last action the thread undoes. If the turn variable is set to this thread, that means that the other thread has given the control to this thread. Since my flag is raised and the turn variable is set, the other thread has to wait in the loop until the current thread is done.
2. Bounded Wait. After one thread lowers, a thread waiting in the while loop will leave because the first condition is broken. This means that threads cannot win all the time.

3. Progress. If no other thread is contesting, other thread's flags are not up. That means that a thread can go past the while loop and do critical section items.

Extra: Implementing Software Mutex

Yes With a bit of searching, it is possible to find it in production for specific simple mobile processors today. Peterson's algorithm is used to implement low-level Linux Kernel locks for the Tegra mobile processor (a system-on-chip ARM process and GPU core by Nvidia) Link to Lock Source

In general now, CPUs and C compilers can re-order CPU instructions or use CPU-core-specific local cache values that are stale if another core updates the shared variables. Thus a simple pseudo-code to C implementation is too naive for most platforms. Warning, here be dragons! Consider this advanced and gnarly topic but (spoiler alert) a happy ending. Consider the following code,

```
while(flag2) {/* busy loop - go around again */
```

An efficient compiler would infer that `flag2` variable is never changed inside the loop, so that test can be optimized to `while(true)` Using `volatile` goes some way to prevent compiler optimizations of this kind.

Let's say that we solved this by telling the compiler not to optimize. Independent instructions can be re-ordered by an optimizing compiler or at runtime by an out-of-order execution optimization by the CPU.

A related challenge is that CPU cores include a data cache to store recently read or modified main memory values. Modified values may not be written back to main memory or re-read from memory immediately. Thus data changes, such as the state of a flag and turn variable in the above example, may not be shared between two CPU codes.

But there is a happy ending. Modern hardware addresses these issues using 'memory fences' also known as a memory barrier. This prevents instructions from getting ordered before or after the barrier. There is a performance loss, but it is needed for correct programs!

Also, there are CPU instructions to ensure that main memory and the CPU's cache is in a reasonable and coherent state. Higher-level synchronization primitives, such as `pthread_mutex_lock` are will call these CPU instructions as part of their implementation. Thus, in practice, surrounding critical sections with a mutex lock and unlock calls is sufficient to ignore these lower-level problems.

For further reading, we suggest the following web post that discusses implementing Peterson's algorithm on an x86 process and the Linux documentation on memory barriers.

1. Memory Fences

2. Memory Barriers

Implementing Counting Semaphore

Now that we have a solution to the critical section problem, We can reasonably implement a mutex. How would we implement other synchronization primitives? Let's start with a semaphore. To implement a semaphore with efficient CPU usage, we will say that we have implemented a condition variable. Implementing an O(1) space condition variable using only a mutex is not trivial, or at least an O(1) heap condition variable is not trivial. We don't want to call malloc while implementing a primitive, or we may deadlock!

- We can implement a counting semaphore using condition variables.
- Each semaphore needs a count, a condition variable and a mutex

```
typedef struct sem_t {
  ssize_t count;
  pthread_mutex_t m;
  pthread_condition_t cv;
} sem_t;
```

Implement sem_init to initialize the mutex and condition variable

```
int sem_init(sem_t *s, int pshared, int value) {
  if (pshared) {
    errno = ENOSYS /* 'Not implemented'*/;
    return -1;
  }

  s->count = value;
  pthread_mutex_init(&s->m, NULL);
  pthread_cond_init(&s->cv, NULL);
  return 0;
}
```

Our implementation of sem_post needs to increment the count. We will also wake up any threads sleeping inside the condition variable. Notice we lock and unlock the mutex so only one thread can be inside the critical section at a time.

```
void sem_post(sem_t *s) {
  pthread_mutex_lock(&s->m);
  s->count++;
  pthread_cond_signal(&s->cv);
  /* A woken thread must acquire the lock, so it will also have to
      wait until we call unlock*/

  pthread_mutex_unlock(&s->m);
}
```

Our implementation of sem_wait may need to sleep if the semaphore's count is zero. Just like sem_post, we wrap the critical section using the lock, so only one thread can be executing our code at a time. Notice if the thread does need to wait then the mutex will be unlocked, allowing another thread to enter sem_post and awaken us from our sleep!

Also notice that even if a thread is woken up before it returns from pthread_cond_wait, it must re-acquire the lock, so it will have to wait until sem_post finishes.

```
void sem_wait(sem_t *s) {
  pthread_mutex_lock(&s->m);
  while (s->count == 0) {
    pthread_cond_wait(&s->cv, &s->m); /*unlock mutex, wait, relock
        mutex*/
  }
  s->count--;
  pthread_mutex_unlock(&s->m);
}
```

That is a complete implementation of a counting semaphore Notice that we are calling sem_post every single time. In practice, this means sem_post would unnecessary call pthread_cond_signal even if there are no waiting threads. A more efficient implementation would only call pthread_cond_signal when necessary i.e.

```
/* Did we increment from zero to one- time to signal a thread
    sleeping inside sem_post */
if (s->count == 1) /* Wake up one waiting thread!*/
pthread_cond_signal(&s->cv);
```

Other semaphore considerations

- A production semaphore implementation may include a queue to ensure fairness and priority. Meaning, we wake up the highest-priority and/or longest sleeping thread.
- An advanced use of sem_init allows semaphores to be shared across processes. Our implementation only works for threads inside the same process. We could fix this by setting the condition variable and mutex attributes.

Extra: Implementing CVs with Mutexes Alone

Implementing a condition variable using only a mutex isn't trivial. Here is a sketch of how we could do it.

```
typedef struct cv_node_ {
  pthread_mutex_t *dynamic;
  int is_awoken;
  struct cv_node_ *next;
} cv_node;
```

```
typedef struct {
  cv_node_ *head
} cond_t

void cond_init(cond_t *cv) {
  cv->head = NULL;
  cv->dynamic = NULL;
}

void cond_destroy(cond_t *cv) {
  // Nothing to see here
  // Though may be useful for the future to put pieces
}

static int remove_from_list(cond_t *cv, cv_node *ptr) {
  // Function assumes mutex is locked
  // Some sanity checking
  if (ptr == NULL) {
    return
  }

  // Special case head
  if (ptr == cv->head) {
    cv->head = cv->head->next;
    return;
  }

  // Otherwise find the node previous
  for (cv_node *prev = cv->head; prev->next; prev = prev->next) {
    // If we've found it, patch it through
    if (prev->next == ptr) {
      prev->next = prev->next->next;
      return;
    }
    // Otherwise keep walking
    prev = prev->next;
  }

  // We couldn't find the node, invalid call

}
```

This is all the boring definitional stuff. The interesting stuff is below.

```
void cond_wait(cond_t *cv, pthread_mutex_t *m) {
  // See note (dynamic) below
  if (cv->dynamic == NULL) {
    cv->dynamic = m
  } else if (cv->dynamic != m) {
    // Error can’t wait with a different mutex!
    abort();
  }
  // mutex is locked so we have the critical section right now
  // Create linked list node _on the stack_
  cv_node my_node;
  my_node.is_awoken = 0;
  my_node.next = cv->head;
  cv->head = my_node.next;
  pthread_mutex_unlock(m);

  // May do some cache busting here
  while(my_node == 0) {
    pthread_yield();
  }

  pthread_mutex_lock(m);
  remove_from_list(cv, &my_node);

  // The dynamic binding is over
  if (cv->head == NULL) {
    cv->dynamic = NULL;
  }
}

void cond_signal(cond_t *cv) {
  for (cv_node *iter = cv->head; iter; iter = iter->next) {
    // Signal makes sure one thread that has not woken up
    // is woken up
    if (iter->is_awoken == 0) {
      // DON’T remove from the linked list here
      // There is no mutual exclusion, so we could
      // have a race condition
      iter->is_awoken = 1;
      return;
    }
  }

  // No more threads to free! No-op
}

void cond_broadcast(cond_t *cv) {
  for (cv_node *iter = cv->head; iter; iter = iter->next) {
    // Wake everyone up!
    iter->is_awoken = 1;
  }
}
```

So how does this work? Instead of allocating space which could lead to deadlock. We keep the data structures or the linked list nodes on each thread's stack. The linked list in the wait function is created **While the thread has the mutex lock** this is important because we may have a race condition on the insert and removal. A more robust implementation would have a mutex per condition variable.

What is the note about (dynamic)? In the pthread man pages, wait creates a runtime binding to a mutex. This means that after the first call is called, a mutex is associated with a condition variable while there is still a thread waiting on that condition variable. Each new thread coming in must have the same mutex, and it must be locked. Hence, the beginning and end of wait (everything besides the while loop) are mutually exclusive. After the last thread leaves, meaning when head is NULL, then the binding is lost.

The signal and broadcast functions merely tell either one thread or all threads respectively that they should be woken up. **It doesn't modify the linked lists because there is no mutex to prevent corruption if two threads call signal or broadcast**

Now an advanced point. Do you see how a broadcast could cause a spurious wakeup in this case? Consider this series of events.

1. Some number more than 2 threads start waiting
2. Another thread calls broadcast.
3. That thread calling broadcast is stopped before it wake any threads.
4. Another thread calls wait on the condition variable and adds itself to the queue.
5. Broadcast iterates through and frees all of the threads.

There is no assurance as to *when* the broadcast was called and when threads were added in a high-performance mutex. The ways to prevent this behavior are to include Lamport timestamps or require that broadcast be called with the mutex in question. That way something that *happens-before* the broadcast call doesn't get signaled after. The same argument is put forward for signal too.

Did you also notice something else? **This is why we ask you to signal or broadcast before you unlock**. If you broadcast after you unlock, the time that broadcast takes could be infinite!

1. Broadcast is called on a waiting queue of threads
2. First thread is freed, broadcast thread is frozen. Since the mutex is unlocked, it locks and continues.
3. It continues for such a long time that it calls broadcast again.
4. With our implementation of a condition variable, this would be terminated. If you had an implementation that appended to the tail of the list and iterated form the head to the tail, this could go on infinitely many times.

In high-performance systems, we want to make sure that each thread that calls wait isn't passed by another thread that calls wait. With the current API that we have, we can't assure that. We'd have to ask users to pass in a mutex or use a global mutex. Instead, we tell programmers to always signal or broadcast before unlocking.

Barriers

Suppose we wanted to perform a multi-threaded calculation that has two stages, but we don’t want to advance to the second stage until the first stage is completed. We could use a synchronization method called a **barrier**. When a thread reaches a barrier, it will wait at the barrier until all the threads reach the barrier, and then they’ll all proceed together.

Think of it like being out for a hike with some friends. You make a mental note of how many friends you have and agree to wait for each other at the top of each hill. Say you’re the first one to reach the top of the first hill. You’ll wait there at the top for your friends. One by one, they’ll arrive at the top, but nobody will continue until the last person in your group arrives. Once they do, you’ll all proceed.

Pthreads has a function pthread_barrier_wait() that implements this. You’ll need to declare a pthread_barrier_t variable and initialize it with pthread_barrier_init(). pthread_barrier_init() takes the number of threads that will be participating in the barrier as an argument. Here is a sample program using barriers.

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>

#define THREAD_COUNT 4

pthread_barrier_t mybarrier;

void* threadFn(void *id_ptr) {
  int thread_id = *(int*)id_ptr;
  int wait_sec = 1 + rand() % 5;
  printf("thread %d: Wait for %d seconds.\n", thread_id, wait_sec);
  sleep(wait_sec);
  printf("thread %d: I’m ready...\n", thread_id);

  pthread_barrier_wait(&mybarrier);

  printf("thread %d: going!\n", thread_id);
  return NULL;
}


int main() {
  int i;
  pthread_t ids[THREAD_COUNT];
  int short_ids[THREAD_COUNT];

  srand(time(NULL));
  pthread_barrier_init(&mybarrier, NULL, THREAD_COUNT + 1);
  for (i=0; i < THREAD_COUNT; i++) {
    short_ids[i] = i;
    pthread_create(&ids[i], NULL, threadFn, &short_ids[i]);
  }

  printf("main() is ready.\n");

  pthread_barrier_wait(&mybarrier);

  printf("main() is going!\n");

  for (i=0; i < THREAD_COUNT; i++) {
    pthread_join(ids[i], NULL);
  }

  pthread_barrier_destroy(&mybarrier);

  return 0;
}
```

Now let's implement our own barrier and use it to keep all the threads in sync in a large calculation. Here is our thought process,

1. Threads do first calculation (use and change values in data)
2. Barrier! Wait for all threads to finish first calculation before continuing
3. Threads do second calculation (use and change values in data)

The thread function has four main parts-

```
// double data[256][8192]

void *calc(void *arg) {
  /* Do my part of the first calculation */
  /* Is this the last thread to finish? If so wake up all the other
      threads! */
  /* Otherwise wait until the other threads have finished part one
      */
  /* Do my part of the second calculation */
}
```

Our main thread will create the 16 threads, and we will divide each calculation into 16 separate pieces. Each thread will be given a unique value (0,1,2,..15), so it can work on its own block. Since a (void*) type can hold small integers, we will pass the value of i by casting it to a void pointer.

```
#define N (16)
double data[256][8192] ;
int main() {
  pthread_t ids[N];
  for(int i = 0; i < N; i++) {
    pthread_create(&ids[i], NULL, calc, (void *) i);
  }
  //...
}
```

Note, we will never dereference this pointer value as an actual memory location.
We will cast it straight back to an integer.

```
void *calc(void *ptr) {
  // Thread 0 will work on rows 0..15, thread 1 on rows 16..31
  int x, y, start = N * (int) ptr;
  int end = start + N;
  for(x = start; x < end; x++) {
    for (y = 0; y < 8192; y++) {
      /* do calc #1 */
    }
  }
}
```

After calculation 1 completes, we need to wait for the slower threads unless we are the last thread! So, keep track of the number of threads that have arrived at our barrier 'checkpoint'.

```
// Global:
int remain = N;

// After calc #1 code:
remain--; // We finished
if (remain == 0) {/*I'm last! - Time for everyone to wake up! */ }
else {
  while (remain != 0) {/* spin spin spin*/ }
}
```

However, the code has a few flaws. One is two threads might try to decrement remain. The other is the loop is a busy loop. We can do better! Let's use a condition variable and then we will use a broadcast/signal functions to wake up the sleeping threads.

A reminder, a condition variable is similar to a house! Threads go there to sleep (pthread_cond_wait). A threa can choose to wake up one thread (pthread_cond_signal) or all of them (pthread_cond_broadcast). If there are no threads currently waiting then these two calls have no effect.

A condition variable version is usually similar to a busy loop incorrect solution - as we will show next. First, let's add a mutex and condition global variables and don't forget to initialize them in main.

```
//global variables
pthread_mutex_t m;
pthread_cond_t cv;

int main() {
  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&cv, NULL);
```

We will use the mutex to ensure that only one thread modifies remain at a time. The last arriving thread needs to wake up *all* sleeping threads - so we will use pthread_cond_broadcast(cv) not pthread_cond_signal

```
pthread_mutex_lock(&m);
remain--;
if (remain == 0) {
  pthread_cond_broadcast(&cv);
}
else {
  while(remain != 0) {
    pthread_cond_wait(&cv, &m);
  }
}
pthread_mutex_unlock(&m);
```

When a thread enters pthread_cond_wait, it releases the mutex and sleeps. After, the thread will be woken up. Once we bring a thread back from its sleep, before returning it must wait until it can lock the mutex. Notice that even if a sleeping thread wakes up early, it will check the while loop condition and re-enter wait if necessary.

The above barrier is not reusable. Meaning that if we stick it into any old calculation loop there is a good chance that the code will encounter a condition where the barrier either deadlocks or thread races ahead one iteration faster. Why is that? Because of the ambitious thread.

We will assume that one thread is much faster than all the other threads. With the barrier API, this thread should be waiting, but it may not be. To make it concrete, let's look at this code

```
void barrier_wait(barrier *b) {
  pthread_mutex_lock(&b->m);
  // If it is 0 before decrement, we should be on
  // another iteration right?
  if (b->remain == 0) b->remain = NUM_THREADS;
  b->remain--;
  if (b->remain == 0) {
    pthread_cond_broadcast(&cv);
  }
  else {
    while(b->remain != 0) {
      pthread_cond_wait(&cv, &m);
    }
  }
  pthread_mutex_unlock(&b->m);
}

for (/* ... */) {
  // Some calc
  barrier_wait(b);
}
```

What happens if a thread becomes ambitious. Well

1. Many other threads wait on the condition variable
2. The last thread broadcasts.
3. A single thread leaves the while loop.
4. This single thread performs its calculation before any other threads *even wake up*
5. Reset the number of remaining threads and goes back to sleep.

All the other threads who should've woken up never do and our implementation deadlocks. How would you go about solving this? Hint: If multiple threads call `barrier_wait` in a loop then one can guarantee that they are on the same iteration.

Reader Writer Problem

Imagine you had a key-value map data structure that is used by many threads. Multiple threads should be able to look up (read) values at the same time provided the data structure is not being written to. The writers are not so gregarious. To avoid data corruption, only one thread at a time may modify (`write`) the data structure and no readers may be reading at that time.

This is an example of the *Reader Writer Problem*. Namely, how can we efficiently synchronize multiple readers and writers such that multiple readers can read together, but a writer gets exclusive access?

An incorrect attempt is shown below ("lock" is a shorthand for `pthread_mutex_lock`):

Attempt #1

```
void read() {
  lock(&m)
  // do read stuff
  unlock(&m)
}

void write() {
  lock(&m)
  // do write stuff
  unlock(&m)
}
```

At least our first attempt does not suffer from data corruption. Readers must wait while a writer is writing and vice versa! However, readers must also wait for other readers. Let's try another implementation.

Attempt #2:

```
void read() {
  while(writing) {/*spin*/}
  reading = 1
  // do read stuff
  reading = 0
}

void write() {
  while(reading || writing) {/*spin*/}
  writing = 1
  // do write stuff
  writing = 0
}
```

Our second attempt suffers from a race condition. Imagine if two threads both called `read` and `write` or both called write at the same time. Both threads would be able to proceed! Secondly, we can have multiple readers and multiple writers, so let's keep track of the total number of readers or writers Which brings us to attempt #3.

Attempt #3

Remember that `pthread_cond_wait` performs *Three* actions. Firstly, it atomically unlocks the mutex and then sleeps (until it is woken by `pthread_cond_signal` or `pthread_cond_broadcast`). Thirdly, the awoken thread must re-acquire the mutex lock before returning. Thus only one thread can actually be running inside the critical section defined by the lock and unlock() methods.

Implementation #3 below ensures that a reader will enter the cond_wait if any writers are writing.

```
read() {
  lock(&m)
  while (writing)
  cond_wait(&cv, &m)
  reading++;

  /* Read here! */

  reading--
  cond_signal(&cv)
  unlock(&m)
}
```

However, only one reader a time can read because candidate #3 did not unlock the mutex. A better version unlocks before reading.

```
read() {
  lock(&m);
  while (writing)
  cond_wait(&cv, &m)
  reading++;
  unlock(&m)

  /* Read here! */

  lock(&m)
  reading--
  cond_signal(&cv)
  unlock(&m)
}
```

Does this mean that a writer and read could read and write at the same time? No! First of all, remember cond_wait requires the thread re-acquire the mutex lock before returning. Thus only one thread can be executing code inside the critical section (marked with **) at a time!

```
read() {
  lock(&m);
  ** while (writing)
  **     cond_wait(&cv, &m)
  ** reading++;
  unlock(&m)
  /* Read here! */
  lock(&m)
  **  reading--
  **  cond_signal(&cv)
  unlock(&m)
}
```

Writers must wait for everyone. Mutual exclusion is assured by the lock.

```
write() {
  lock(&m);
  **  while (reading || writing)
  **      cond_wait(&cv, &m);
  **  writing++;
  **
  ** /* Write here! */
  **  writing--;
  **  cond_signal(&cv);
  unlock(&m);
}
```

Candidate #3 above also uses `pthread_cond_signal`. This will only wake up one thread. If many readers are waiting for the writer to complete, only one sleeping reader will be awoken from their slumber. The reader and writer should use `cond_broadcast` so that all threads should wake up and check their while-loop condition.

Starving writers

Candidate #3 above suffers from starvation. If readers are constantly arriving then a writer will never be able to proceed (the 'reading' count never reduces to zero). This is known as *starvation* and would be discovered under heavy loads. Our fix is to implement a bounded-wait for the writer. If a writer arrives they will still need to wait for existing readers however future readers must be placed in a "holding pen" and wait for the writer to finish. The "holding pen" can be implemented using a variable and a condition variable so that we can wake up the threads once the writer has finished.

The plan is that when a writer arrives, and before waiting for current readers to finish, register our intent to write by incrementing a counter 'writer'

```
write() {
  lock()
  writer++

  while (reading || writing)
  cond_wait
  unlock()
  ...
}
```

And incoming readers will not be allowed to continue while writer is nonzero. Notice 'writer' indicates a writer has arrived, while 'reading' and 'writing' counters indicate there is an *active* reader or writer.

```
read() {
  lock()
  // readers that arrive *after* the writer arrived will have to
     wait here!
  while(writer)
  cond_wait(&cv,&m)

  // readers that arrive while there is an active writer
  // will also wait.
  while (writing)
  cond_wait(&cv,&m)
  reading++
  unlock
  ...
}
```

Attempt #4

Below is our first working solution to the Reader-Writer problem. Note if you continue to read about the "Reader Writer problem" then you will discover that we solved the "Second Reader Writer problem" by giving writers preferential access to the lock. This solution is not optimal. However, it satisfies our original problem of N active readers, single active writer, and avoiding starvation of the writer if there is a constant stream of readers.

Can you identify any improvements? For example, how would you improve the code so that we only woke up readers or one writer?

```
int writers; // Number writer threads that want to enter the
    critical section (some or all of these may be blocked)
int writing; // Number of threads that are actually writing inside
    the C.S. (can only be zero or one)
int reading; // Number of threads that are actually reading inside
    the C.S.
// if writing !=0 then reading must be zero (and vice versa)

reader() {
  lock(&m)
  while (writers)
  cond_wait(&turn, &m)
  // No need to wait while(writing here) because we can only exit
     the above loop
  // when writing is zero
  reading++
  unlock(&m)

  // perform reading here

  lock(&m)
  reading--
  cond_broadcast(&turn)
  unlock(&m)
}

writer() {
  lock(&m)
  writers++
  while (reading || writing)
  cond_wait(&turn, &m)
  writing++
  unlock(&m)
  // perform writing here
  lock(&m)
  writing--
  writers--
  cond_broadcast(&turn)
  unlock(&m)
}
```

Ring Buffer

A ring buffer is a simple, usually fixed-sized, storage mechanism where contiguous memory is treated as if it is circular, and two index counters keep track of the current beginning and end of the queue. As array indexing is not circular, the index counters must wrap around to zero when moved past the end of the array. As data is added (enqueued) to the front of the queue or removed (dequeued) from the tail of the queue, the current items in the buffer form a train that appears to circle the track

A simple (single-threaded) implementation is shown below. Note, enqueue and dequeue do not guard against underflow or overflow. It's possible to add an item when the queue is full and possible to remove an item when the queue is empty. If we added 20 integers (1, 2, 3, . . . , 20) to the queue and did not dequeue any items then values, `17,18,19,20` would overwrite the `1,2,3,4`. We won't fix this problem right now, instead of when we create the multi-threaded version we will ensure enqueue-ing and dequeue-ing threads are blocked while the ring buffer is full or empty respectively.

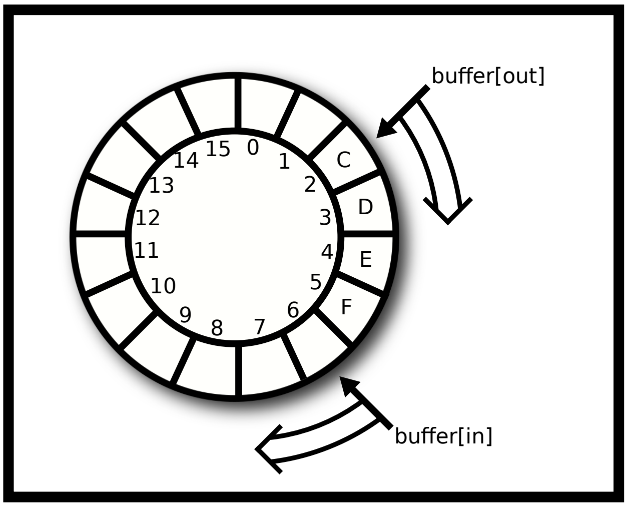


Figure 7.1: Ring Buffer Visualization

```
void *buffer[16];
unsigned int in = 0, out = 0;

void enqueue(void *value) {/* Add one item to the front of the
    queue*/
  buffer[in] = value;
  in++; /* Advance the index for next time */
  if (in == 16) in = 0; /* Wrap around! */
}

void *dequeue() {/* Remove one item to the end of the queue.*/
  void *result = buffer[out];
  out++;
  if (out == 16) out = 0;
  return result;
}
```

Ring Buffer Gotchas

It's tempting to write the enqueue or dequeue method in the following compact form.

```
// N is the capacity of the buffer
void enqueue(void *value)
b[ (in++) % N ] = value;
}
```

This method would appear to work but contains a subtle bug. With more than four billion enqueue operations, the int value of in will overflow and wrap around to 0! Thus, you might end up writing into b[0] for example!

A compact form is correct uses bit masking provided N is a power of two. (16,32,64,...)

```
b[ (in++) & (N-1) ] = value;
```

This buffer does not yet prevent overwrites. For that, we'll turn to our multi-threaded attempt that will block a thread until there is space or there is at least one item to remove.

Multithreaded Correctness

The following code is an incorrect implementation. What will happen? Will enqueue and/or dequeue block? Is mutual exclusion satisfied? Can the buffer underflow? Can the buffer overflow? For clarity, pthread_mutex is shortened to p_m and we assume sem_wait cannot be interrupted.

```
#define N 16
void *b[N]
int in = 0, out = 0
p_m_t lock
sem_t s1,s2
void init() {
  p_m_init(&lock, NULL)
  sem_init(&s1, 0, 16)
  sem_init(&s2, 0, 0)
}

enqueue(void *value) {
  p_m_lock(&lock)

  // Hint: Wait while zero. Decrement and return
  sem_wait( &s1 )

  b[ (in++) & (N-1) ] = value

  // Hint: Increment. Will wake up a waiting thread
  sem_post(&s1)
  p_m_unlock(&lock)
}
void *dequeue(){
  p_m_lock(&lock)
  sem_wait(&s2)
  void *result = b[(out++) & (N-1) ]
  sem_post(&s2)
  p_m_unlock(&lock)
  return result
}
```

Analysis

Before reading on, see how many mistakes you can find. Then determine what would happen if threads called the enqueue and dequeue methods.

- The enqueue method waits and posts on the same semaphore (s1) and similarly with enqueue and (s2) i.e. we decrement the value and then immediately increment the value, so by the end of the function the semaphore value is unchanged!
- The initial value of s1 is 16, so the semaphore will never be reduced to zero - enqueue will not block if the ring buffer is full - so overflow is possible.
- The initial value of s2 is zero, so calls to dequeue will always block and never return!
- The order of mutex lock and sem_wait will need to be swapped; however, this example is so broken that this bug has no effect!

Another Analysis

The following code is an incorrect implementation. What will happen? Will enqueue and/or dequeue block? Is mutual exclusion satisfied? Can the buffer underflow? Can the buffer overflow? For clarity pthread_mutex is shortened to p_m and we assume sem_wait cannot be interrupted.

```
void *b[16]
int in = 0, out = 0
p_m_t lock
sem_t s1, s2
void init() {
  sem_init(&s1,0,16)
  sem_init(&s2,0,0)
}

enqueue(void *value){
  sem_wait(&s2)
  p_m_lock(&lock)

  b[ (in++) & (N-1) ] = value

  p_m_unlock(&lock)
  sem_post(&s1)
}
```

```
void *dequeue(){
  sem_wait(&s1)
  p_m_lock(&lock)
  void *result = b[(out++) & (N-1)]
  p_m_unlock(&lock)
  sem_post(&s2)

  return result;
}
```

Here are a few problems that we hope you've found.

- The initial value of s2 is 0. Thus enqueue will block on the first call to sem_wait even though the buffer is empty!
- The initial value of s1 is 16. Thus dequeue will not block on the first call to sem_wait even though the buffer is empty - Underflow! The dequeue method will return invalid data.
- The code does not satisfy Mutual Exclusion. Two threads can modify `in` or `out` at the same time! The code appears to use mutex lock. Unfortunately, the lock was never initialized with `pthread_mutex_init()` or `PTHREAD_MUTEX_INITIALIZER` - so the lock may not work (`pthread_mutex_lock` may simply do nothing)

Correct implementation of a ring buffer

As the mutex lock is stored in global (static) memory it can be initialized with `PTHREAD_MUTEX_INITIALIZER`. If we had allocated space for the mutex on the heap, then we would have used `pthread_mutex_init(ptr, NULL)`

```
#include <pthread.h>
#include <semaphore.h>
// N must be 2^i
#define N (16)

void *b[N]
int in = 0, out = 0
p_m_t lock = PTHREAD_MUTEX_INITIALIZER
sem_t countsem, spacesem

void init() {
  sem_init(&countsem, 0, 0)
  sem_init(&spacesem, 0, 16)
}
```

The enqueue method is shown below. Make sure to note.

1. The lock is only held during the critical section (access to the data structure).

2. A complete implementation would need to guard against early returns from `sem_wait` due to POSIX signals.

```
enqueue(void *value){
  // wait if there is no space left:
  sem_wait( &spacesem )

  p_m_lock(&lock)
  b[ (in++) & (N-1) ] = value
  p_m_unlock(&lock)

  // increment the count of the number of items
  sem_post(&countsem)
}
```

The `dequeue` implementation is shown below. Notice the symmetry of the synchronization calls to `enqueue`. In both cases, the functions first wait if the count of spaces or count of items is zero.

```
void *dequeue(){
  // Wait if there are no items in the buffer
  sem_wait(&countsem)

  p_m_lock(&lock)
  void *result = b[(out++) & (N-1)]
  p_m_unlock(&lock)

  // Increment the count of the number of spaces
  sem_post(&spacesem)

  return result
}
```

Food for thought:

- What would happen if the order of `pthread_mutex_unlock` and `sem_post` calls were swapped?
- What would happen if the order of `sem_wait` and `pthread_mutex_lock` calls were swapped?

Extra: Process Synchronization

You thought that you were using different processes, so you don't have to synchronize? Think again! You may not have race conditions within a process but what if your process needs to interact with the system around it? Let's consider a motivating example

```c
void write_string(const char *data) {
  int fd = open("my_file.txt", O_WRONLY);
  write(fd, data, strlen(data));
  close(fd);
}

int main() {
  if(!fork()) {
    write_string("key1: value1");
    wait(NULL);
  } else {
    write_string("key2: value2");
  }
  return 0;
}
```

If none of the system calls fail then we should get something that looks like this given the file was empty to begin with.

```
key1: value1
key2: value2
```

```
key2: value2
key1: value1
```

Interruption

But, there is a hidden nuance. Most system calls can be `interrupted` meaning that the operating system can stop an ongoing system call because it needs to stop the process. So barring `fork` `wait` `open` and `close` from failing – they typically go to completion – what happens if `write` fails? If write fails and no bytes are written, we can get something like `key1: value1` or `key2: value2`. This is data loss which is incorrect but won't corrupt the file. What happens if write gets interrupted after a partial write? We get all sorts of madness. For example,

```
key2: key1: value1
```

Solution

A program can create a mutex before fork-ing - however the child and parent process will not share virtual memory and each one will have a mutex independent of the other. Advanced note: There are advanced options using shared memory that allow a child and parent to share a mutex if it's created with the correct options and uses a shared memory segment. See stackoverflow example

So what should we do? We should use a shared mutex! Consider the following code.

```
pthread_mutex_t * mutex = NULL;
pthread_mutexattr_t attr;

void write_string(const char *data) {
  pthread_mutex_lock(mutex);
  int fd = open("my_file.txt", O_WRONLY);
  int bytes_to_write = strlen(data), written = 0;
  while(written < bytes_to_write) {
    written += write(fd, data + written, bytes_to_write - written);
  }
  close(fd);
  pthread_mutex_unlock(mutex);
}

int main() {
  pthread_mutexattr_init(&attr);
  pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
  pmutex = mmap (NULL, sizeof(pthread_mutex_t),
  PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, -1, 0);
  pthread_mutex_init(pmutex, &attrmutex);
  if(!fork()) {
    write_string("key1: value1");
    wait(NULL);
    pthread_mutex_destroy(pmutex);
    pthread_mutexattr_destroy(&attrmutex);
    munmap((void *)pmutex, sizeof(*pmutex));
  } else {
    write_string("key2: value2");
  }
  return 0;
}
```

What the code does in main is initialize a process shared mutex using a piece of shared memory. You will find out what this call to mmap does later – assume for the time being that it creates memory that is shared between processes. We can initialize a pthread_mutex_t in that special piece of memory and use it as normal. To counter write failing, we have put the write call inside a while loop that keeps writing so long as there are bytes left to write. Now if all the other system calls function, there should be more race conditions.

Most programs try to avoid this problem entirely by writing to separate files, but it is good to know that there are mutexes across processes, and they are useful. A program can use all of the primitives that were mentioned previouslty! Barriers, semaphores, and condition variables can all be initialized on a shared piece of memory and used in similar ways to their multithreading counterparts.

- You don't have to worry about arbitrary memory addresses becoming race condition candidates. Only areas that specifically mapped are in danger.
- You get the nice isolation of processes so if one process fails the system can maintain intact.
- When you have a lot of threads, creating a process might ease the system load

Extra: Higher Order Models of Synchronization

When using atomics, you need to specify the right model of synchronization to ensure a program behaves correctly. You can read more about them On the gcc wiki These examples are adapted from those.

Sequentially Consistent

Sequentially consistent is the simplest, least error-prone and most expensive model. This model says that any change that happens, all changes before it will be synchronized between all threads.

```
Thread 1                     Thread 2
1.0 atomic_store(x, 1)
1.1 y = 10                   2.1 if (atomic_load(x) == 0)
1.2 atomic_store(x, 0);      2.2    y != 10 && abort();
```

Will never quit. This is because either the store happens before the if statement in thread 2 and y == 1 or the store happens after and x does not equal 2.

Relaxed

Relaxed is a simple memory order providing for more optimizations. This means that only a particular operation needs to be atomic. One can have stale reads and writes, but after reading the new value, it won't become old.

```
-Thread 1-               -Thread 2-
atomic_store(x, 1);      printf("%d\n", x) // 1
atomic_store(x, 0);      printf("%d\n", x) // could be 1 or 0
                         printf("%d\n", x) // could be 1 or 0
```

But that means that previous loads and stores don't need to affect other threads. In the previous example, the code can now fail.

Acquire/Release

The order of atomic variables don't need to be consistent – meaning if atomic var y is assigned to 10 then atomic var x to be 0 those don't need to propagate, and a threa could get stale reads. Non-atomic variables have to get updated in all threads though.

Consume

Imagine the same as above except non-atomic variables don't need to get updated in all threads. This model was introduced so that there can be an Acquire/Release/Consume model without mixing in Relaxed because Consume is similar to relax.

Extra: Actor Model and Goroutines

There are a *lot* of other methods of concurrency than described in this book. Posix threads are the finest grained thread construct, allowing for tight control of the threads and the CPU. Other languages have their abstractions. We'll talk about a language go that is similar to C in terms of simplicity and design, go or golang To get the 5 minute introduction, feel free to read the learn x in y guide for go. Here is how we create a "thread" in go.

```
func hello(out) {
    fmt.Println(out);
}

func main() {
    to_print := "Hello World!"
    go hello(to_print)
}
```

This actually creates what is known as a goroutine. A goroutine can be thought of as a lightweight thread. Internally, it is a worker pool of threads that executes instructions of all the running goroutines. When a goroutine needs to be stopped, it is frozen and "context switched" to another thread. Context switch is in quotes because this is done at the run time level versus real context switching which is done at the operating system level.

The advantage to gofuncs is pretty self explanatory. There is no boilerplate code, or joining, or odd casting `void *`.

We can still use mutexes in go to perform our end result. Consider the counting example as before.

```
var counter = 0;
var mut sync.Mutex;
var wg sync.WaitGroup;

func plus() {
  mut.Lock()
  counter += 1
  mut.Unlock()
  wg.Done()
}

func main() {
  num := 10
  wg.Add(num);
  for i := 0; i < num; i++ {
    go plus()
  }

  wg.Wait()

  fmt.Printf("%d\n", counter);

}
```

But that's boring and error prone. Instead, let's use the actor model. Let's designate two actors. One is the main actor that will be performing the main instruction set. The other actor will be the counter. The counter is responsible for adding numbers to an internal variable. We'll send messages between the threads when we want to add and see the value.

```
const (
  addRequest = iota;
  outputRequest = iota;
)

func counterActor(requestChannel chan int, outputChannel chan int)
    {
  counter := 0

  for {
    req := <- requestChannel;
    if req == addRequest {
      counter += 1
    } else if req == outputRequest {
      outputChannel <- counter
    }
  }
}

func main() {
  // Set up the actor
  requestChannel := make(chan int)
  outputChannel := make(chan int)
  go counterActor(requestChannel, outputChannel)

  num := 10
  for i := 0; i < num; i++ {
    requestChannel <- addRequest
  }
  requestChannel <- outputRequest
  new_count := <- outputChannel
  fmt.Printf("%d\n", new_count);
}
```

Although there is a bit more boilerplate code, we don't have mutexes anymore! If we wanted to scale this operation and do other things like increment by a number, or write to a file, we can have that particular actor take care of it. This differentiation of responsibilities is important to make sure your design scales well. There are even libraries that handle all of the boilerplate code as well.

External Resources

Guiding questions for the man pages

- How is a recursive mutex different than a default mutex?
- How is mutex trylock different than mutex lock?
- Why would a mutex lock fail? What's an example?
- What happens if a thread tries to destroy a locked mutex?
- Can a thread copy the underlying bytes of a mutex instead of using a pointer?
- What is the lifecycle of a semaphore?

- pthread_mutex_lock man page
- pthread_mutex_init man page
- sem_init
- sem_wait
- sem_post
- sem_destroy

Topics

- Atomic operations
- Critical Section
- Producer Consumer Problem
- Using Condition Variables
- Using Counting Semaphore
- Implementing a barrier

- Implementing a ring buffer
- Using pthread_mutex
- Implementing producer consumer
- Analyzing multi-threaded coded

Questions

- What is atomic operation?
- Why will the following not work in parallel code

```
//In the global section
size_t a;
//In pthread function
for(int i = 0; i < 100000000; i++) a++;
```

And this will?

```
//In the global section
atomic_size_t a;
//In pthread function
for(int i = 0; i < 100000000; i++) atomic_fetch_add(a, 1);
```

- What are some downsides to atomic operations? What would be faster: keeping a local variable or many atomic operations?
- What is the critical section?
- Once you have identified a critical section, what is one way of assuring that only one thread will be in the section at a time?
- Identify the critical section here

```
struct linked_list;
struct node;
void add_linked_list(linked_list *ll, void* elem){
  node* packaged = new_node(elem);
  if(ll->head){
    ll->head =
  }else{
    packaged->next = ll->head;
    ll->head = packaged;
    ll->size++;
  }
}

void* pop_elem(linked_list *ll, size_t index){
  if(index >= ll->size) return NULL;

  node *i, *prev;
  for(i = ll->head; i && index; i = i->next, index--){
    prev = i;
  }

  //i points to the element we need to pop, prev before
  if(prev->next) prev->next = prev->next->next;
  ll->size--;
  void* elem = i->elem;
  destroy_node(i);
  return elem;
}
```

- How tight can you make the critical section?
- What is a producer consumer problem? How might the above be a producer consumer problem be used in the above section? How is a producer consumer problem related to a reader writer problem?
- What is a condition variable? Why is there an advantage to using one over a `while` loop?
- Why is this code dangerous?

```
if(not_ready){
  pthread_cond_wait(&cv, &mtx);
}
```

- What is a counting semaphore? Give me an analogy to a cookie jar/pizza box/limited food item.
- What is a thread barrier?
- Use a counting semaphore to implement a barrier.
- Write up a Producer/Consumer queue, How about a producer consumer stack?
- Give me an implementation of a reader-writer lock with condition variables, make a struct with whatever you need, it needs to be able to support the following functions

```
typedef struct {


} rw_lock_t;

void reader_lock(rw_lock_t* lck) {

}

void writer_lock(rw_lock_t* lck) {

}

void reader_unlock(rw_lock_t* lck) {

}

void writer_unlock(rw_lock_t* lck) {

}
```

The only specification is that in between reader_lock and reader_unlock, no writers can write. In between the writer locks, only one writer may be writing at a time.

- Write code to implement a producer consumer using ONLY three counting semaphores. Assume there can be more than one thread calling enqueue and dequeue. Determine the initial value of each semaphore.
- Write code to implement a producer consumer using condition variables and a mutex. Assume there can be more than one thread calling enqueue and dequeue.
- Use CVs to implement add(unsigned int) and subtract(unsigned int) blocking functions that never allow the global value to be greater than 100.
- Use CVs to implement a barrier for 15 threads.
- What does the following code do?

```
void main() {
    pthread_mutex_t mutex;
    pthread_cond_t cond;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    pthread_cond_broadcast(&cond);
    pthread_cond_wait(&cond,&mutex);

    return 0;
}
```

- Is the following code correct? If it isn't, could you fix it?

```
extern int money;
void deposit(int amount) {
  pthread_mutex_lock(&m);
  money += amount;
  pthread_mutex_unlock(&m);
}

void withdraw(int amount) {
  if (money < amount) {
    pthread_cond_wait(&cv);
  }

  pthread_mutex_lock(&m);
  money -= amount;
  pthread_mutex_unlock(&m);
}
```

- Sketch how to use a binary semaphore as a mutex. Remember in addition to mutual exclusion, a mutex can only ever be unlocked by the thread who called it.

```
sem_t sem;

void lock() {

}

void unlock() {

}
```

- How many of the following statements are true?
  - There can be multiple active readers
  - There can be multiple active writers
  - When there is an active writer the number of active readers must be zero
  - If there is an active reader the number of active writers must be zero
  - A writer must wait until the current active readers have finished

Bibliography

[1] T.J. Dekker and Edsgar Dijkstra. Over de sequentialiteit van procesbeschrijvingen, 1965. URL `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD00xx/EWD35.html`.

[2] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12:115–116, 1981.

8 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 sometimes you find
You get what you need

The philosophers Jagger & Richards

Deadlock is defined as when a system cannot make and forward progress. We define a system for the rest of the chapter as a set of rules by which a set of processes can move from one state to another, where a state is either working or waiting for a particular resource. Forward progress is defined as if there is at least one process working or we can award a process waiting for a resource that resource. In a lot of systems, Deadlock is avoided by ignoring the entire concept [5, P.237]. Have you heard about turn it on and off again? For products where the stakes are low (User Operating Systems, Phones), it may be more efficient to allow deadlock. But in the cases where "failure is not an option" - Apollo 13, you need a system that tracks, breaks, or prevents deadlocks. Apollo 13 didn't failed because of deadlock, but it wouldn't be good to restart the system on liftoff.

Mission-critical operating systems need this guarantee 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 the higher the chance the method will work.

Resource Allocation Graphs

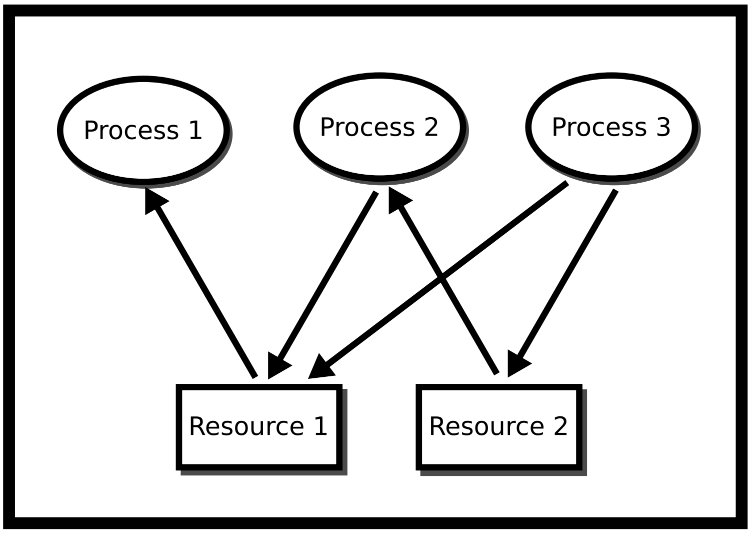


Figure 8.1: Resource allocation graph

One such way is modeling the system with a resource allocation graph (RAG). 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 a simple yet powerful 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 will be deadlocked 8.1. We'll make the distinction that the system is deadlocked by definition if all workers cannot perform an operation other than waiting. We can detect a deadlock by traversing the graph and searching for a cycle using a graph traversal algorithm, such as the Depth First Search (DFS). This graph is considered as a directed graph and we can treat both the processes and resources as nodes.

m

```
typedef struct {
    int node_id; // Node in this particular graph
    Graph **reachable_nodes; // List of nodes that can be reached
        from this node
    int size_reachable_nodes; // Size of the List
} Graph;

// isCyclic() traverses a graph using DFS and detects whether it
    has a cycle
// isCyclic() uses a recursive approach
// G points to a node in a graph, can be either a resource or a
    process
```

```
// is_visited is an array indexed with node_id and initialized
    with zeros (false) to record whether a particular node has been
    visited
int isCyclic(Graph *G, int* is_visited) {
    if (this graph has been visited) {
        // Oh! the cycle is found
        return true;
    } else {
        1. Mark this node as visited
        2. Traverse through all nodes in the reachable_nodes
        3. Call isCyclic() for each node
        4. Evaluate the return value of isCyclic()
    }
    // Nope, this graph is acyclic
    return false;
}
```

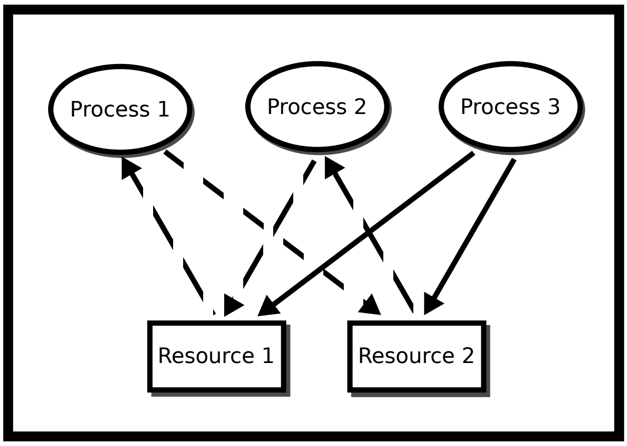


Figure 8.2: Graph based Deadlock

Coffman Conditions

Surely cycles in RAGs happen all the time in an OS, so why doesn't it grind to a halt? You may not see deadlock because the OS may **preempt** some processes breaking the cycle but there is still a chance that your three lonely processes could deadlock.

There are four *necessary* and *sufficient* conditions for deadlock – meaning if these conditions hold then there is a non-zero probability that the system will deadlock at any given iteration. These are known as the Coffman Conditions [2].

- 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 {P1,P2,…} such that P1 is waiting for resources held by P2, which is waiting for P3,…, which is waiting for P1.
- Hold and Wait: Once a resource is obtained, a process keeps the resource locked.
- No Pre-emption: Nothing can force the process to give up a resource.

Proof: Deadlock can happen if and only if the four Coffman conditions are satisfied.

$\rightarrow$ If the system is deadlocked, the four Coffman conditions are apparent.

- For the purposes of contradiction, assume that there is no circular wait. If not then that means the resource allocation graph is acyclic, meaning that there is at least one process that is not waiting on any resource to be freed. Since the system can move forward, the system is not deadlocked.
- For the purposes of contradiction, assume that there is no mutual exclusion. If not, that means that no process is waiting on any other process for a resource. This breaks circular wait and the previous argument proves correctness.
- For the purposes of contradiction, assume that processes don't hold and wait but our system still deadlocks. Since we have circular wait from the first condition at least one process must be waiting on another process. If that and processes don't hold and wait, that means one process must let go of a resource. Since the system has moved forward, it cannot be deadlocked.
- For the purposes of contradiction, assume that we have preemption, but the system cannot be undeadlocked. Have one process, or create one process, that recognizes the circular wait that must be apparent from above and break one the of the links. By the first branch, we must not have deadlocked.

$\leftarrow$ If the four conditions are apparent, the system is deadlocked. We will prove that if the system is not deadlocked, the four conditions are not apparent. Though this proof is not formal, let us build a system with the three requirements not including circular wait. Let assume that there is a set of processes $P = \{p_1, p_2, ..., p_n\}$ and there is a set of resources $R = \{r_1, r_2, ..., r_m\}$. For simplicity, a process can only request one resource at a time but the proof can be generalized to multiple. Let assume that the system is a state at time t. Let us assume that the state of the system is a tuple (h_t, w_t) where there are two function $h_t : R \rightarrow P \cup \{\text{unassigned}\}$ that maps resources to the processes that own them (this is a function, meaning that we have mutual exclusion) and or unassigned and $w_t : P \rightarrow R \cup \{\text{satisfied}\}$ that maps the requests that each process makes to a resource or if the process is satisfied. If the process is satisfied, we consider the work trivial and the process exits, releasing all resources – this can also be generalized. Let $L_t \subseteq P \times R$ be a set of list of requests that a process uses to release a resource at any given time. The evolution of the system is at each step at every time.

- Release all resources in L_t.
- Find a process that is requesting a resource
- If that resource is available give it to that process, generating a new (h_{t+1}, w_{t+1}) and exit the current iteration.
- Else find another process and try the same resource allocation procedure in the previous step.

If all processes have been surveyed and if all are requesting a resource and none can be granted a resource, consider it deadlocked. More formally, this system is deadlocked means if $\exists t_0, \forall t \geq t_0, \forall p \in P, w_t(p) \neq$ satisfied and $\exists q, q \neq p \rightarrow h_t(w_t(p)) = q$ (which is what we need to prove).

Mutual exclusion and no pre-emption are encoded into the system. Circular wait implies the second condition, a resource is owned by another process which is owned by another process meaning at this state $\forall p \in P, \exists q \neq p \rightarrow h_t(w_t(p)) = q$. Circular wait also implies that at this current state, no process is satisfied, meaning at this state $\forall p \in P, w_t(p) \neq$ satisfied. Hold and wait simply proves the condition that from this point onward, the system will not change, which is all the conditions that we needed to show. □

If a system breaks any of them, it 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 fails 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 a held item or tell both students to put the items down.

Livelock relates to deadlock. Consider the breaking hold and wait solution as above. Though deadlock is avoided, if the philosopher pick up the same device again and again in the exact same pattern, no work will be 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 fails to 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 each of the steps of a system and if each of the steps eventually – after some finite number of steps – leads to forward progress, the system fails to livelock. There are even better systems that prove bounded waits; a system can only be livelocked for at most n cycles which may be important for something like stock exchanges.

Approaches to Solving Livelock and Deadlock

Ignoring deadlock is the most obvious approach. 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 any deadlock usually goes away. An operating system preempts processes when stopping them for context switches. The operating system can 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 the OS deadlocks. 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 to break deadlock. As an example, consider multiple processes accessing files. The operating system can 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 [4] 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 systems are good for products that need to maintain a non-deadlocked state but can tolerate a small chance of livelock for a short time.

Extra: Banker's Algorithm

We can start out with a single resource Banker's Algorithm. Consider a banker. The banker has a finite amount of money. With a finite amount of money, she want to make loans and eventually get my money back. Let's say that we have a set of n people where each of them has a set amount or a limit a_i (i being the ith process) that they need to obtain before they can do any work. The banker keep track in my book how much she has given to each person l_i, and she has some amount of moeny p at any given time. For people to request money, they do the following. Consider the state of the system $(A = \{a_1, a_2, ...\}, L_t = \{l_{t,1}, l_{t,2}, ...\}, p)$ at time t. A precondition is that we have $p \geq min(A)$, or we have enough money to suit at least one person.. Also, each person will work for a finite period and give back our money.

- A person j requests m from me
  - if $m < p$, they are denied.
  - if $m + l_j > a_i$ they are denied
  - Pretend we are in a new state $(A, L_{t+1} = \{.., l_{t+1,j} = l_{t,j} + m, ...\}, p - m)$ where the process is granted the resource.
- if now person j is either satisfied ($l_{t+1,j} == a_j$) or $min(a_i - l_{t+1,i}) \leq p$. In other words, we have enough money to suit one other person. If either, consider the transaction safe and give them the money.

Why does this work? Well at the start we are in a safe state – defined by we have enough money to suit at least one person. Each of these "loans" results in a safe state. If we have exhausted our reserve, one person is working and will give us money greater than or equal to our previous "loan", thus putting us in a safe state again. Since we can always make one additional move, the system can never deadlock. Now, there is no guarantee that the system won't livelock. If the process we hope to request something never does, no work will be done – but not due to deadlock. This analogy expands to higher orders of magnitude but requires that either a process can do its work entirely or there exists a process whose combination of resources can be satisfied, which makes the algorithm a little more tricky (an additional for loop) but nothing too bad. There are some notable downsides.

- The program first needs to know how much of each resource a process needs. A lot of times that is impossible or the process requests the wrong amount because the programmer didn't foresee it.
- The system could livelock.
- We know in most systems that resources vary, pipes and sockets for example. This could mean that the runtime of the algorithm could be slow for systems with millions of resources.

- Also, this can't keep track of the resources that come and go. A process may delete a resource as a side effect or create a resource. The algorithm assumes a static allocation and that each process performs a non-destructive operation.

Dining Philosophers

The Dining Philosophers problem is a classic synchronization problem. Imagine we invite *n* (let's say 6) philosophers to a meal. We will sit them at a table with 6 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.

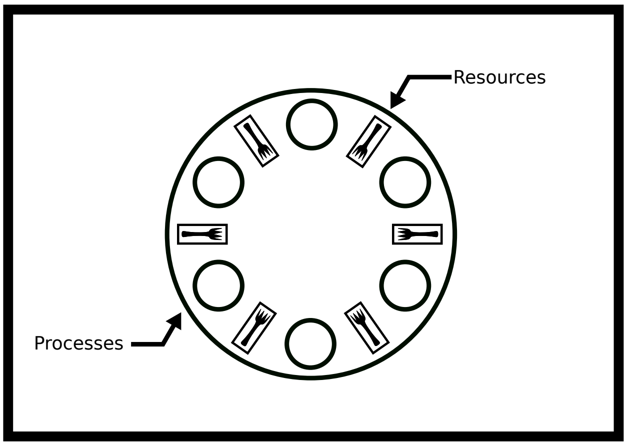


Figure 8.3: Dining Philosophers

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 is essentially the same, meaning that each philosopher has the same instruction set based on the other philosopher i.e. 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 deadlock goes down as the number of philosophers goes up. What is important to note is that eventually that this solution will deadlock, letting threads starve which is bad. Here is a simple resource allocation graph that shows how the system could be deadlocked

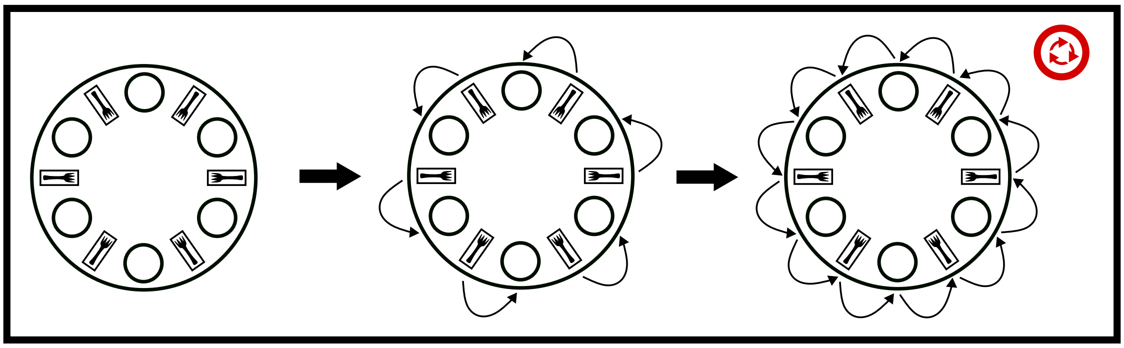


Figure 8.4: Left right dining philosopher cycle

So now you are thinking about breaking one of the Coffman Conditions. 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){
    int left_succeed = pthread_mutex_trylock(left_fork);
    if (!left_succeed) {
      sleep();
      continue;
    }
    int right_succeed = pthread_mutex_trylock(right_fork);
    if (!right_succeed) {
      pthread_mutex_unlock(left_fork);
      sleep();
      continue;
    }
    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 and so on. Here is what a time evolution of the system would look like.

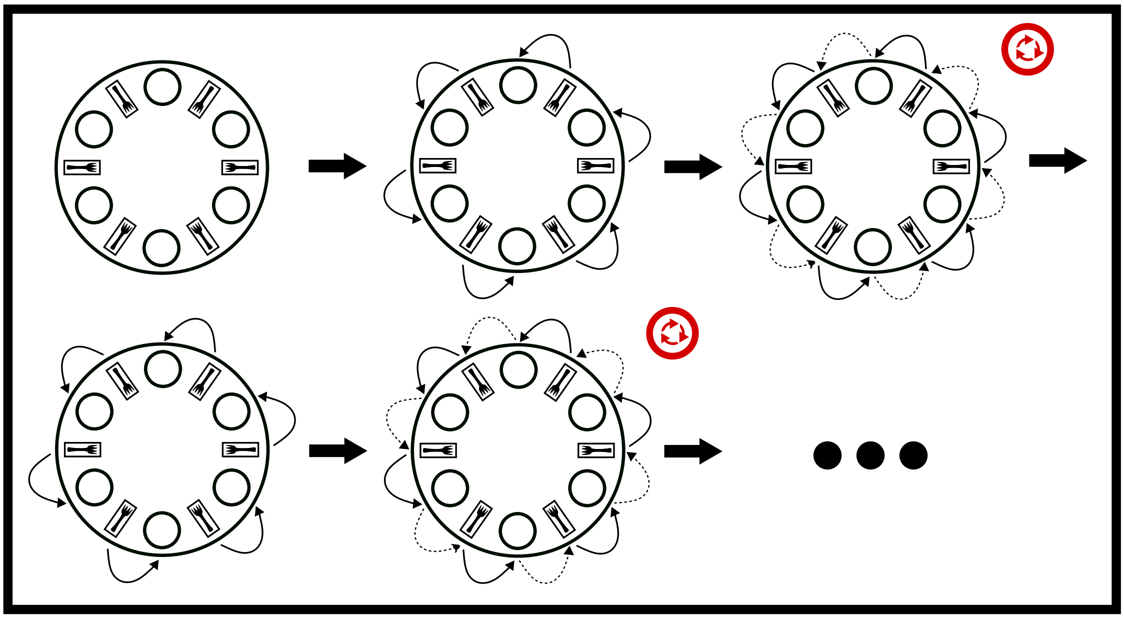


Figure 8.5: Livelock Failure

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 has one arbitrator a mutex for example. Have each of the philosophers ask the arbitrator for permission to eat or trylock an arbitrator 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 for 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 philosophers 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. Assuming that all the philosophers are good-willed, the arbitrator needs to be fair. In practical systems, the arbitrator tends to give forks to the same processes because of scheduling or pseudo-randomness. 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 Descartes 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.

Proof: The arbitrator solution doesn't deadlock

The proof is about as simple as it gets. Only one philosopher can request resources at a time. There is no way to make a cycle in the resource allocation graph with only one philosopher acting in pickup the left then the right fork which is what we needed to show.

□

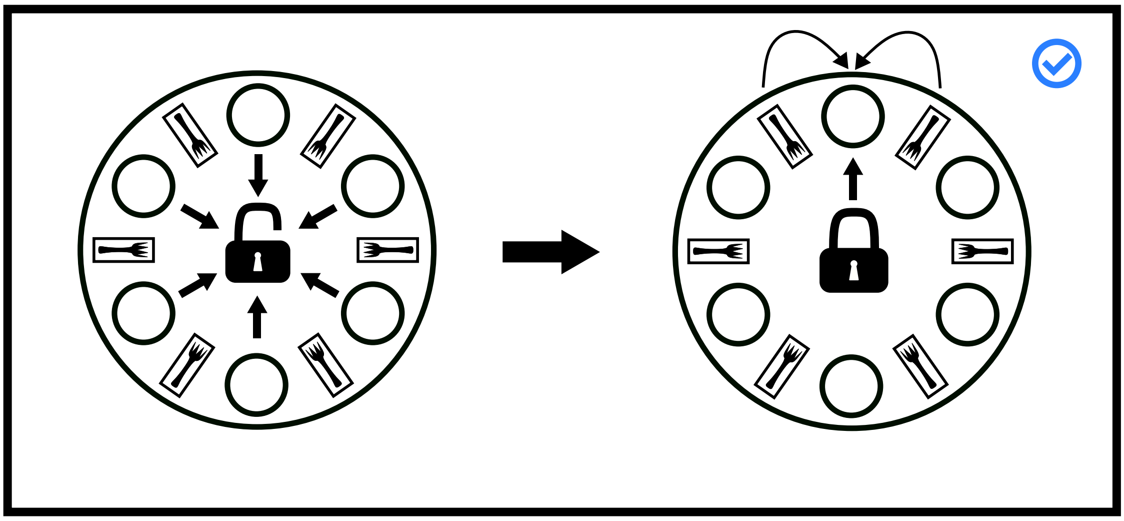


Figure 8.6: Arbitrator Diagram

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' [6, P. 280] solution removes 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 philosophers 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 beforehand. In the dining philosophers case, this is solved because everything is known but trying to specify an operating system where a system doesn't know which file is going to get opened by what process can lead to 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 to ensure bounded wait times.

Proof: Stallings' Solution Doesn't Deadlock. Let's number the philosophers $\{p_0, p_1, .., p_{n-1}\}$ and the resources $\{r_0, r_1, .., r_{n-1}\}$. A philosopher p_i needs resource $r_{i-1 \mod n}$ and $r_{i+1 \mod n}$. Without loss of generality, let us take p_i out of the picture. Each resource had exactly two philosophers that could use it. Now resources $r_{i-1 \mod n}$ and $r_{i+1 \mod n}$ only have on philosopher waiting on it. Even if hold and wait, no preemption, and mutual exclusion or present, the resources can never enter a state where one philosopher requests them and they are held by another philosopher because only one philosopher can request them. Since there is no way to generate a cycle otherwise, circular wait cannot hold. Since circular wait cannot hold, deadlock cannot happen. □

Here is a visualization of the worst-case. The system is about to deadlock, but the approach resolves it.

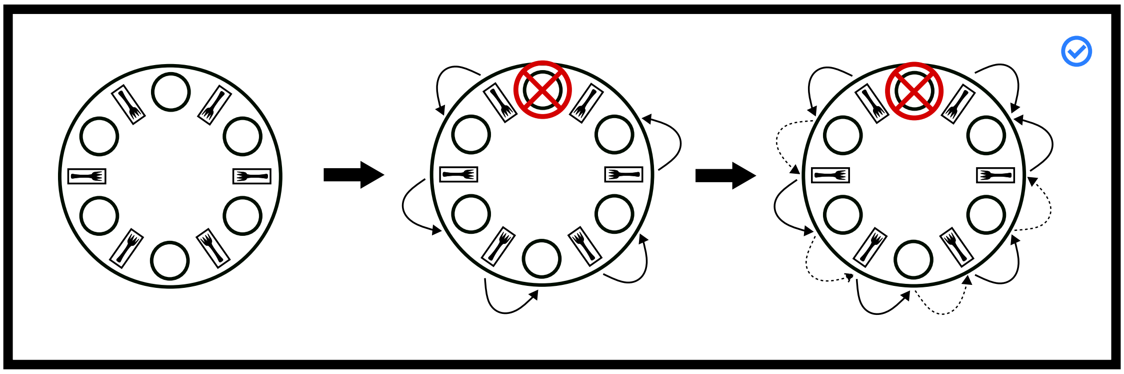


Figure 8.7: Stalling solution almost deadlock

Partial Ordering (Dijkstra's Solution)

This is Dijkstra's solution [3, P. 20]. 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 philosophers 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.

Some problems are that an entity either needs to know the finite set of resources in advance or be able to produce a consistent partial order suck that circular wait cannot happen. This also implies that there needs to be some entity, either the operating system or another process, deciding on the number and all of the philosophers need to agree on the number as new resources come in. As we have also seen with previous solutions, this relies on context switching. This prioritizes philosophers that have already eaten but can be made fairer by introducing random sleeps and waits.

Proof: Dijkstra's Solution Doesn't Deadlock

The proof is similar to the previous proof. Let's number the philosophers $\{p_0, p_1, .., p_{n-1}\}$ and the resources $\{r_0, r_1, .., r_{n-1}\}$. A philosopher p_i needs resource $r_{i-1 \mod n}$ and $r_{i+1 \mod n}$. Each philosopher will grab $r_{i-1 \mod n}$ then $r_{i+1 \mod n}$ but the last philosopher will grab in the reverse order. Even if hold and wait, no preemption, and mutual exclusion or present. Since the last philosopher will grab r_{n-1} then r_0 there are two cases either the philosopher has the first lock or the philosopher doesn't.

If the last philosopher p_{n-1} holds the first lock meaning the previous philosopher p_{n-2} is waiting on r_{n-1} meaning r_{n-2} is available. Since no other blockers, the philosopher previous p_{n-3} will grab her first lock. This is now a reduction to the previous proof of stalling because we now have n resources but only $n-1$ philosophers, meaning this cannot deadlock.

If the philosopher doesn't obtain that first lock, then we have a reduction to Stalling's proof above because now have $n-1$ philosophers vying for n resources. Since we can't reach deadlock in either case, this solution cannot deadlock which is what we needed to show.

□

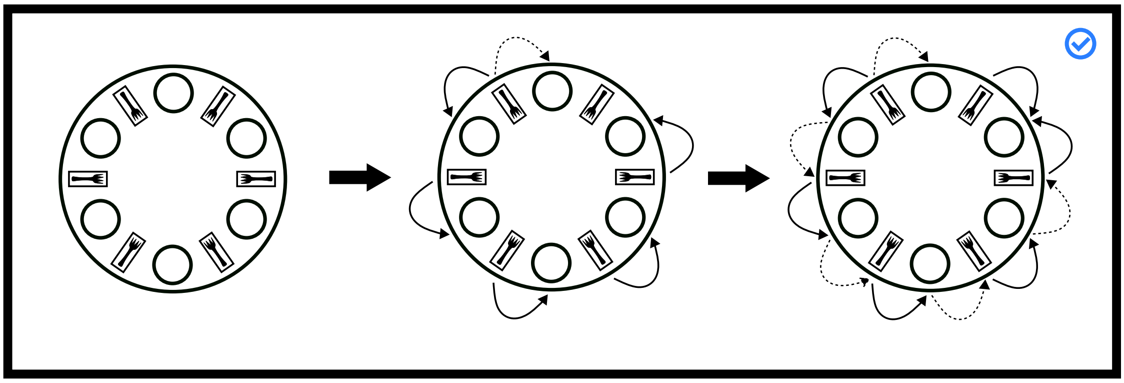


Figure 8.8: Stalling solution partial deadlock

Extra: Clean/Dirty Forks (Chandy/Misra Solution)

There are many more advanced solutions. One such solution is by Chandy and Misra [1]. This is not a true solution to the dining philosophers problem because it has the requirement that philosophers can speak to each other. It is a solution that ensures fairness for some notion of fairness. In essence, it defines a series of rounds that a philosopher must eat in a given round before going to the next one.

We won't detail the proof here because it is a little more involved, but feel free to read more.

Extra: Actor Model

The actor model is another form of synchronization that doesn't have to do anything with negotiating locks or waiting. The idea is simple. Each actor can either perform work, create more actors, send messages, or respond to messages. Any time an actor needs something from another actor, it sends a message. Most importantly, an actor is only responsible for one thing. If we were implementing a real-world application, we may have an actor that handles the database, one that handles the incoming connections, one that services the connections, etc etc. These actors would pass messages to each other like "there is a new connection" from the incoming connection actor to the servicing actor. The servicing actor may send a data request message the database actor and a data response message comes back.

While this seems like the perfect solution there are drawbacks. The first is the actual library of communication needs to be synchronized. If you don't have a framework that does this already – like the Message Passing Interface or MPI for High-Performance Computing – then the framework will have to be built and would most likely be as much work to build efficiently compared to direct synchronization. Also, the messages now encounter additional overhead for serializing and deserializing or at the least. And a final drawback is that an actor could take an arbitrarily long time to respond to a message, spurring the need for shadow actors who service the same job.

As mentioned, there are frameworks like Message passing interface that is somewhat based on the actor model and allows distributed systems in high performance computing to work effectively, but your mileage may vary If you want to read further on the model, feel free to glance over the wikipedia page listed below. Further reading on the actor model

Topics

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

Questions

- What are the Coffman Conditions?
- What do each of the Coffman conditions mean? Provide a definition of each one.
- Give a real-life example of breaking each Coffman condition in turn. A situation to consider: Painters, Paint, Paint Brushes etc. How would you assure that work would get done?
- Which Coffman condition is unsatisfied in the following snippet?

```
// 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 can complete if it can 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

[1] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4): 632–646, October 1984. ISSN 0164-0925. doi: 10.1145/1780.1804. URL `http://doi.acm.org/10.1145/1780.1804`.

[2] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[3] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. published as [**?**]WD:EWD310pub, n.d. URL `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF`.

[4] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. ISSN 00029947. URL `http://www.jstor.org/stable/1990888`.

[5] 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=WjvX0HmVTlMC`.

[6] William Stallings. *Operating Systems: Internals and Design Principles 7th Ed. by Stallings (International Economy Edition)*. PE, 2011. ISBN 9332518807. URL https://www.amazon.com/Operating-Systems-Internals-Principles-International/dp/9332518807?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=9332518807.

9

Virtual Memory and Interprocess Communication

Abbott: Now you've got it.
Costello: I throw the ball to Naturally.
Abbott: You don't! You throw it to Who!
Costello: Naturally.
Abbott: Well, that's it - say it that way.
Costello: That's what I said.

Abbott and Costello on Effective Communication

In simple embedded systems and early computers, processes directly access memory – "Address 1234" corresponds to a particular byte stored in a particular part of physical memory. For example, the IBM 709 had to read and write directly to tape with no level of abstraction [3, P. 65]. Even in systems after that, it was hard to adopt virtual memory because virtual memory required the whole fetch cycle to be altered through hardware – a change many manufacturers still thought was expensive. In the PDP-10, a workaround was used by using different registers for each process and then virtual memory was added later [1]. In modern systems, this is no longer the case. Instead, each process is isolated, and there is a translation process between the address of a particular CPU instruction or piece of data of a process and the actual byte of physical memory ("RAM"). Memory addresses no longer map to physical addresses The process runs inside virtual memory. Virtual memory keeps processes safe because one process cannot directly read or modify another process's memory. Virtual memory also allows the system to efficiently allocate and reallocate portions of memory to different processes. The modern process of translating memory is as follows.

1. A process makes a memory request
2. The circuit first checks the Translation Lookaside Buffer (TLB) if the address page is cached into memory. It skips to the reading from/writing to phase if found otherwise the request goes to the MMU.
3. The Memory Management Unit (MMU) performs the address translation. If the translation succeeds, the page gets pulled from RAM – conceptually the entire page isn't loaded up. The result is cached in the TLB.
4. The CPU performs the operation by either reading from the physical address or writing to the address.

Translating Addresses

The Memory Management Unit is part of the CPU, and it converts a virtual memory address into a physical address. First, we'll talk about what the virtual memory abstraction is and how to translate addresses

To illustrate, consider a 32-bit machine, meaning pointers are 32-bits. They can address 2^{32} different locations or 4GB of memory where one address is one byte. Imagine we had a large table for every possible address where we will store the 'real' i.e. physical address. Each physical address will need 4 bytes – to hold the 32-bits. Naturally, This scheme would require 16 billion bytes to store all of the entries. It should be painfully obvious that our lookup scheme would consume all of the memory that we could buy for our 4GB machine. Our lookup table should be smaller than the memory we have otherwise we will have no space left for our actual programs and operating system data. The solution is to chunk memory into small regions called 'pages' and 'frames' and use a lookup table for each page.

Terminology

A **page** is a block of virtual memory. A typical block size on Linux is 4KiB or 2^{12} addresses, though one can find examples of larger blocks. So rather than talking about individual bytes, we can talk about blocks of 4KiBs, each block is called a page. We can also number our pages ("Page 0" "Page 1" etc). Let's do a sample calculation of how many pages are there assume page size of 4KiB.

For a 32-bit machine,

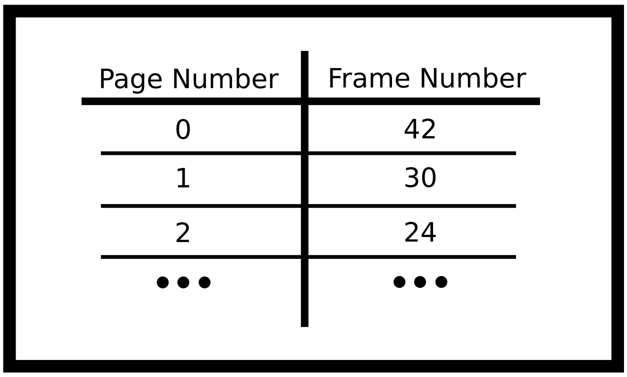
$$2^{32}\text{address}/2^{12}(\text{address/page}) = 2^{20}\text{pages}.$$

For a 64-bit machine,

$$2^{64}\text{address}/2^{12}(\text{address/page}) = 2^{52}\text{pages} \approx 10^{15}\text{pages}.$$

We also call this a **frame** or sometimes called a 'page frame' is a block of *physical memory* or RAM – Random Access Memory. A frame is the same number of bytes as a virtual page or 4KiB on our machine. It stores the bytes of interest. To access a particular byte in a frame, an MMU goes from the start of the frame and adds the offset – discussed later.

A **page table** is a map from a number to a particular frame. For example Page 1 might be mapped to frame 45, page 2 mapped to frame 30. Other frames might be currently unused or assigned to other running processes or used internally by the operating system. Implied from the name, imagine a page table as a table.



Page Number	Frame Number
0	42
1	30
2	24
•••	•••

Figure 9.1: Explicit Frame Table

In practice, we will omit the first column because it will always be sequentially 0, 1, 2, etc and instead we'll use the offset from the start of the table as the entry number.

Now to go through the actual calculations. We will assume that a 32-bit machine has 4KiB pages Naturally, to address all the possible entries, there are 2^{20} frames. Since there are 2^{20} possible frames, we will need 20 bits to number all of the possible frames meaning `Frame Number` must be 2.5 bytes long. In practice, we'll round that up to 4 bytes and do something interesting with the rest of the bits. With 4 bytes per entry x 2^{20} entries = 4 MiB of physical memory are required to hold the entire page table for a process.

Remember our page table maps pages to frames, but each frame is a block of contiguous addresses. How do we calculate which particular byte to use inside a particular frame? The solution is to re-use the lowest bits of the virtual memory address directly. For example, suppose our process is reading the following address- `VirtualAddress = 11110000111100001111000010101010 (binary)`

So to give an example say we have the virtual address above. How would we split it up using a one-page table to frame scheme?

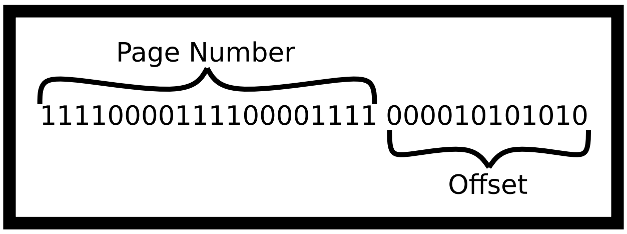


Figure 9.2: Splitting Address

We can imagine the steps to dereference as one process. In general, it looks like the following.

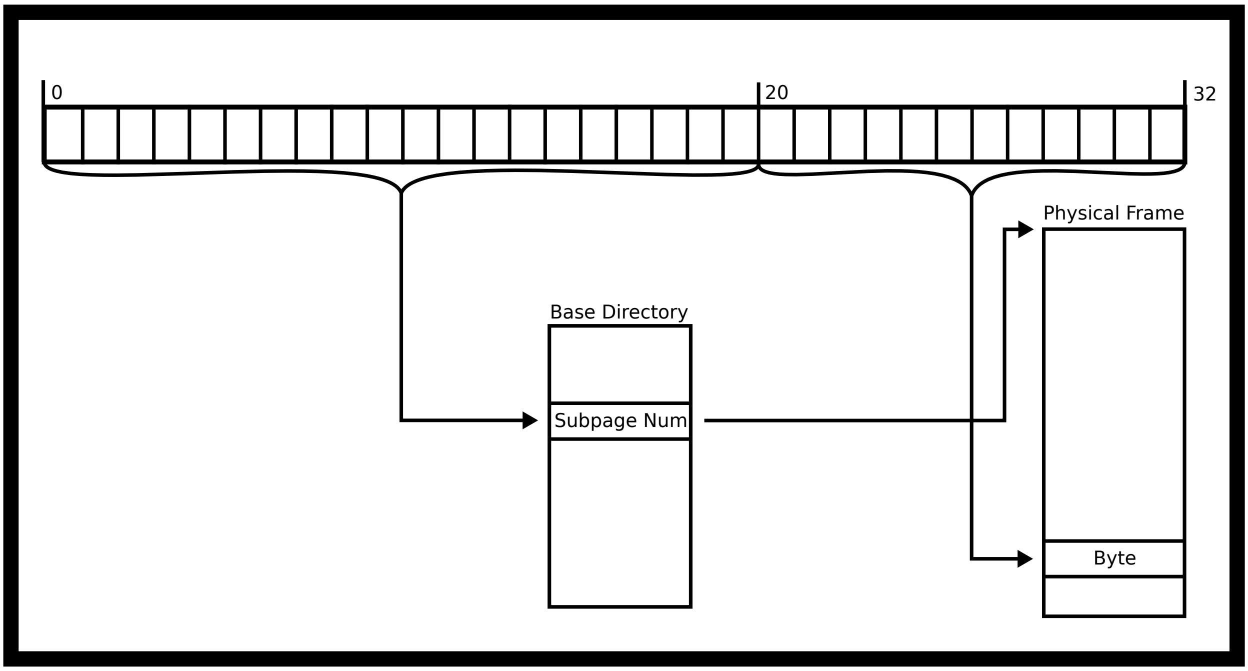


Figure 9.3: One level dereference

The way to read from a particular address above is visualized below.

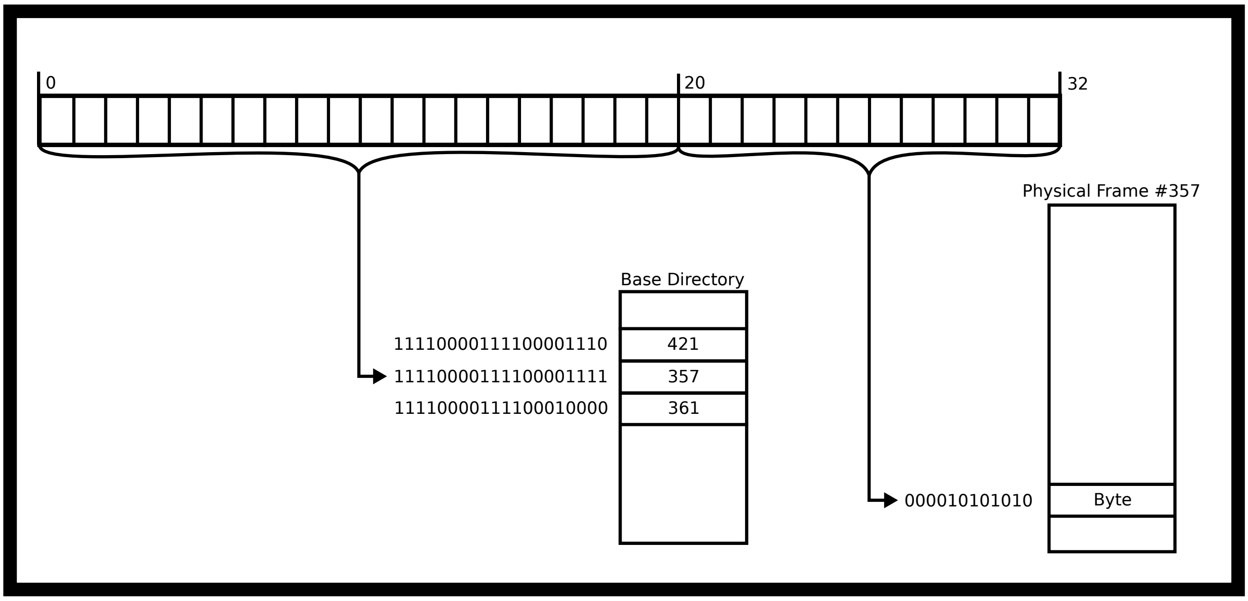


Figure 9.4: One level dereference example

And if we were reading from it, 'return' that value. This sounds like a perfect solution. Take each address and map it to a virtual address in sequential order. The process will believe that the address looks continuous, but the top 20 bits are used to figure out `page_num`, which will allow us to find the frame number, find the frame, add the **offset** – derived from the last 12 bits – and do the read or write.

There are other ways to split it as well. On a machine with page size 256 Bytes, then the lowest 8 bits (10101010) will be used as the offset. The remaining upper bits will be the page number (111100001111000011110000). This offset is treated as a binary number and is added to the start of the frame when we get it.

We do have a problem with 64-bit operating systems. For a 64-bit machine with 4KiB pages, each entry needs 52 bits. Meaning we need roughly With 2^{52} entries, that's 2^{55} bytes (roughly 40 petabytes). So our page table is too large. In 64-bit architecture, memory addresses are sparse, so we need a mechanism to reduce the page table size, given that most of the entries will never be used. We'll take about this below. There is one last piece of terminology that needs to be covered.

Multi-level page tables

Multi-level pages are one solution to the page table size issue for 64-bit architectures. We'll look at the simplest implementation - a two-level page table. Each table is a list of pointers that point to the next level of tables, som sub-tables may be omitted. An example, a two-level page table for a 32-bit architecture is shown below.

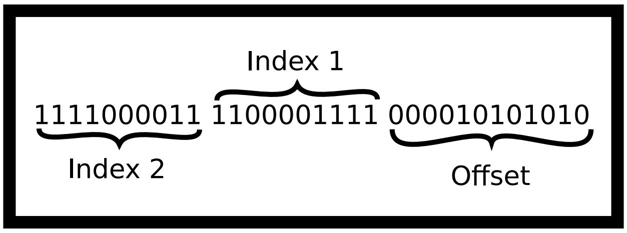


Figure 9.5: Three Way Address Split

So what is the intuition for dereferencing an address? First, the MMU takes the top-level page table and find the `Index1`'th entry. That will contain a number that will lead the MMU to the appropriate sub-table Then go to the `Index2`'th entry of that table. That will contain a frame number. This is the good old fashioned 4KiB RAM that we were talking about earlier. Then, the MMU adds the offset and do the read or write.

Visualizing The Dereference

In one diagram, the dereference looks like the following image.

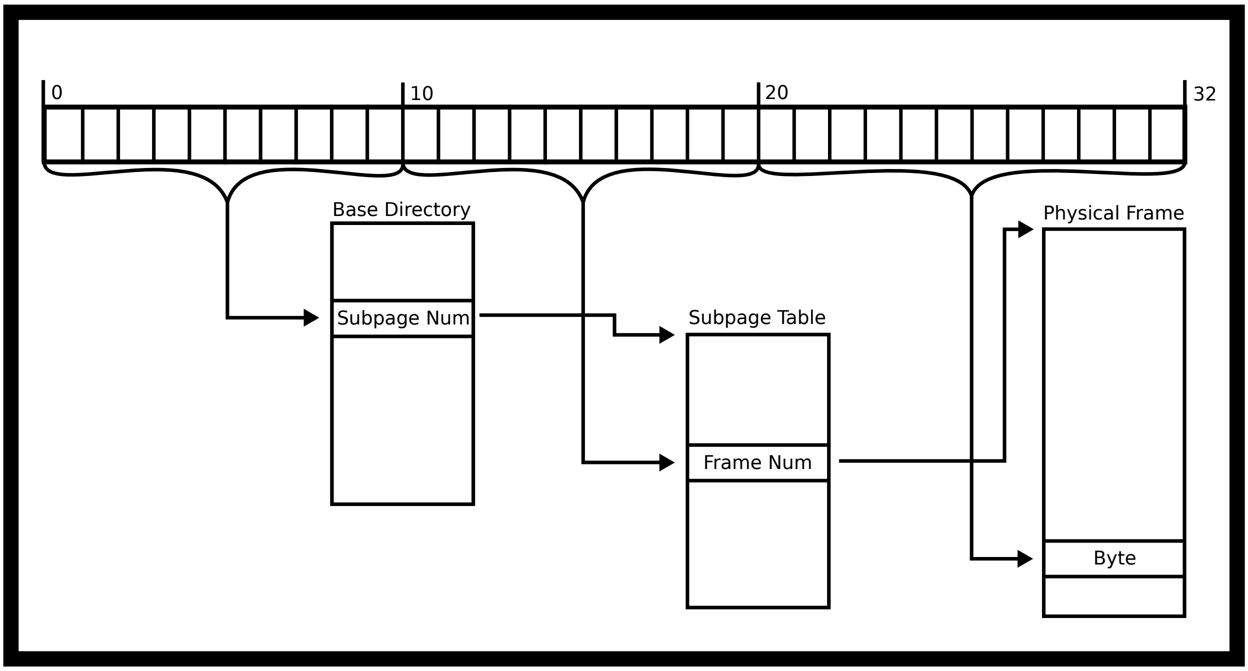


Figure 9.6: Full page table dereference

Following our example, here is what the dereference would look like.

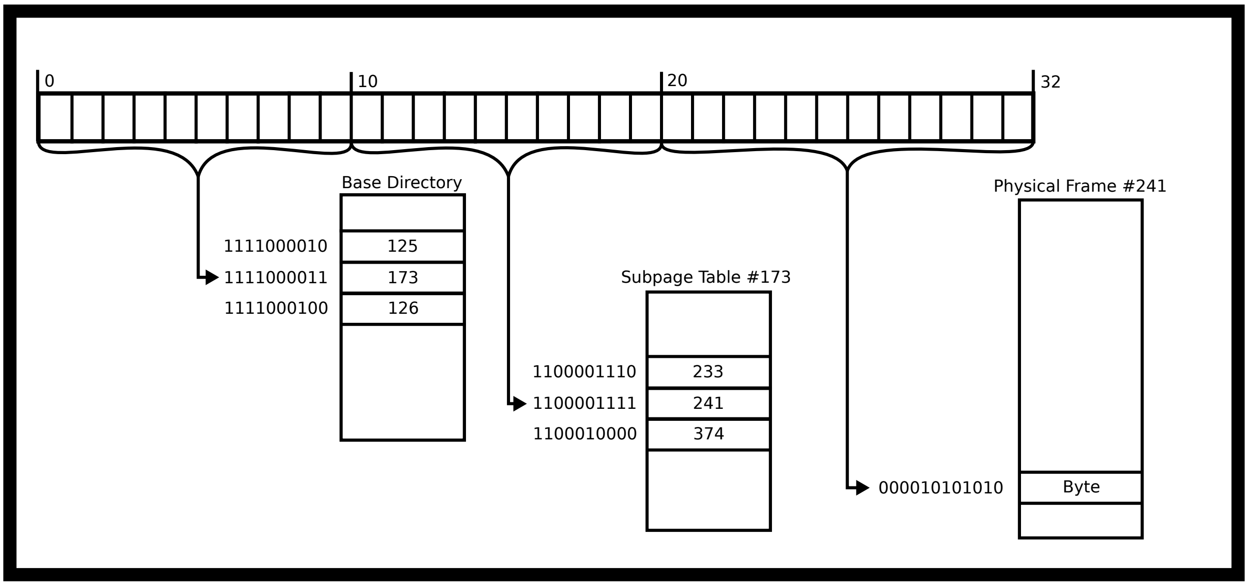


Figure 9.7: Full page example dereference

Calculating Size Concerns

Now some calculations on size. Each `page_table_num` index is 10 bits wide because there are only 2^{10} possible `sub-tables`, so we need 10 bits to store each directory index. We'll round up to 2 bytes for the sake of reasoning. If 2 bytes are used for each entry in the top-level table and there are only 2^{10} entries, we only need 2KiB to store this entire first level page table. Each subtable will point to physical frames, and each of their entries needs to be the required 4 bytes to be able to address all the frames as mentioned earlier. However, for processes with only tiny memory needs, we only need to specify entries for low memory addresses for the heap and program code and high memory addresses for the stack.

Thus, the total memory overhead for our multi-level page table has shrunk from 4MiB for the single-level implementation to three page tables of memory or 2KiB for the top-level and 4KiB for the two intermediate levels of size 10KiB. Here's why. We need at least one frame for the high-level directory and two frames for two sub-tables. One sub-table is necessary for the low addresses – program code, constants and possibly a tiny heap. The other sub-table is for higher addresses used by the environment and stack. In practice, real programs will likely need more sub-table entries, as each subtable can only reference 1024*4KiB = 4MiB of address space. The main point still stands. We have significantly reduced the memory overhead required to perform page table lookups.

Page Table Disadvantages

There are lots of problems with page tables – one of the big problems is that they are slow. For a single page table, our machine is now twice as slow! Two memory accesses are required. For a two-level page table, memory access is now three times as slow – three memory accesses are required.

To overcome this overhead, the MMU includes an associative cache of recently-used virtual-page-to-frame lookups. This cache is called the TLB ("translation lookaside buffer"). Every time a virtual address needs to be translated into a physical memory location, the TLB is queried in parallel to the page table. For most memory accesses of most programs, there is a significant chance that the TLB has cached the results. However, if a program has inadequate cache coherence, the address will be missing in the TLB, meaning the MMU must use the much slower page table translation.

MMU Algorithm

There is a sort of pseudocode associated with the MMU. We will assume that this is for a single-level page table.

1. Receive address
2. Try to translate address according to the programmed scheme
3. If the translation fails, report an invalid address
4. Otherwise,
    (a) If the TLB contains the physical memory, get the physical frame from the TLB and perform the read and write.
    (b) If the page exists in memory, check if the process has permissions to perform the operation on the page meaning the process has access to the page, and it is reading from the page/writing to a page that it has permission to do so.
        i. If so then do the dereference provide the address, cache the results in the TLB
        ii. Otherwise, trigger a hardware interrupt. The kernel will most likely send a SIGSEGV or a Segmentation Violation.
    (c) If the page doesn't exist in memory, generate an Interrupt.
        i. The kernel could realize that this page could either be not allocated or on disk. If it fits the mapping, allocate the page and try the operation again.
        ii. Otherwise, this is invalid access and the kernel will most likely send a SIGSEGV to the process.

How would you alter this for a multi-level page table?

Frames and Page Protections

Frames can be shared between processes, and this is where the heart of the chapter comes into play. We can use these tables to communicate with processes. In addition to storing the frame number, the page table can be used to store whether a process can write or only read a particular frame. Read-only frames can then be safely shared between multiple processes. For example, the C-library instruction code can be shared between all processes that dynamically load the code into the process memory. Each process can only read that memory. Meaning that if a program tries to write to a read-only page in memory, it will `SEGFAULT`. That is why sometimes memory accesses SEGFAULT and sometimes they don't, it all depends on if your hardware says that a program can access.

Also, processes can share a page with a child process using the `mmap` system call. `mmap` is an interesting call because instead of tying each virtual address to a physical frame, it ties it to something else. It is an important distinction that we are talking about mmap and not memory-mapped IO in general. The `mmap` system call can't reliably be used to do other memory-mapped operations like communicate with GPUs and write pixels to the screen – this is mainly hardware dependent.

Bits on Pages

This is *heavily* dependent on the chipset. We will include some bits that have historically been popular in chipsets.

1. The read-only bit marks the page as read-only. Attempts to write to the page will cause a page fault. The page fault will then be handled by the Kernel. Two examples of the read-only page include sharing the C standard library between multiple processes for security you wouldn't want to allow one process to modify the library and Copy-On-Write where the cost of duplicating a page can be delayed until the first write occurs.

2. The execution bit defines whether bytes in a page can be executed as CPU instructions. Processors may merge these bits into one and deem a page either writable or executable. This bit is useful because it prevents stack overflow or code injection attacks when writing user data into the heap or the stack because those are not read-only and thus not executable. Further reading: background

3. The dirty bit allows for performance optimization. A page exclusively read from can be discarded without syncing to disk, since the page hasn't changed. However, if the page was written to after it's paged in, its dirty bit will be set, indicating that the page must be written back to the backing store. This strategy requires that the backing store retain a copy of the page after it is paged into memory. When a dirty bit is omitted, the backing store need only be as large as the instantaneous total size of all paged-out pages at any moment. When a dirty bit is used, at all times some pages will exist in both physical memory and the backing store.

4. There are plenty of other bits. Take a look at your favorite architecture and see what other bits are associated!

Page Faults

A page fault may happen when a process accesses an address in a frame missing in memory. There are three types of Page Faults

1. **Minor** If there is no mapping yet for the page, but it is a valid address. This could be memory asked for by `sbrk(2)` but not written to yet meaning that the operating system can wait for the first write before allocating space – if it was read from, the operating system could short circuit the operation to read 0. The OS simply makes the page, loads it into memory, and moves on.

2. **Major** If the mapping to the page is exclusively on disk. The operating system will swap the page into memory and swap another page out. If this happens frequently enough, your program is said to *thrash* the MMU.

3. **Invalid** When a program tries to write to a non-writable memory address or read to a non-readable memory address. The MMU generates an invalid fault and the OS will usually generate a `SIGSEGV` meaning segmentation violation meaning that the program wrote outside the segment that it could write to.

Link Back to IPC

What does this have to do with IPC? Before, you knew that processes had isolation. One, you didn't know how that isolation mapped. Two, you may not know how you can break this isolation. To break any memory level isolation you have two avenues. One is to ask the kernel to provide some kind of interface. The other is to ask the kernel to map two pages of memory to the same virtual memory area and handle all the synchronization yourself.

mmap

`mmap` is a trick of virtual memory of instead of mapping a page to a frame, that frame can be backed by a file on disk, or the frame can be shared among processes. We can use that to read from a file on disk efficiently or sync changes to the file. One of the big optimizations is a file may be lazily allocated to memory. Take the following code for example.

```
int fd = open(...); //File is 2 Pages
char* addr = mmap(..fd..);
addr[0] = 'l';
```

The kernel sees that the program wants to mmap the file into memory, so it will reserve some space in your address space that is the length of the file. That means when the program writes to `addr[0]` that it writes to the first byte of the file. The kernel can do some optimizations too. Instead of loading the whole file into memory, it may only load pages at a time. A program may only access 3 or 4 pages making loading the entire file a waste of time. Page faults are so powerful because let the operating system take control of when a file is used.

mmap Definitions

`mmap` does more than take a file and map it to memory. It is the general interface for creating shared memory among processes. Currently it only supports regular files and POSIX `shmem` [2]. Naturally, you can read all about it in the reference above, which references the current working group POSIX standard. Some other options to note in the page will follow.

The first option is that the `flags` argument of mmap can take many options.

1. `PROT_READ` This means the process can read the memory. This isn't the only flag that gives the process read permission, however! The underlying file descriptor, in this case, must be opened with read privileges.

2. `PROT_WRITE` This means the process can write to the memory. This has to be supplied for a process to write to a mapping. If this is supplied and `PROT_NONE` is also supplied, the latter wins and no writes can be performed. The underlying file descriptor, in this case, must either be opened with write privileges or a private mapping must be supplied below

3. `PROT_EXEC` This means the process can execute this piece of memory. Although this is not stated in POSIX documents, this shouldn't be supplied with WRITE or NONE because that would make this invalid under the NX bit or not being able to execute (respectively)

4. `PROT_NONE` This means the process can't do anything with the mapping. This could be useful if you implement guard pages in terms of security. If you surround critical data with many more pages that can't be accessed, that decreases the chance of various attacks.

5. `MAP_SHARED` This mapping will be synchronized to the underlying file object. The file descriptor must've been opened with write permissions in this case.

6. `MAP_PRIVATE` This mapping will only be visible to the process itself. Useful to not thrash the operating system.

Remember that once a program is done mmapping that the program must munmap to tell the operating system that it is no longer using the pages allocated, so the OS can write it back to disk and give back the addresses in case another mmap needs to occur. There are accompanying calls msync that take a piece of mmap'ed memory and sync the changes back to the filesystem though we won't cover that in-depth. The other parameters to mmap are described in the annotated walkthrough below.

Annotated mmap Walkthrough

Below is an annotated walkthrough of the example code in the man pages. Our command-line utility will take a file, offset, and length to print. We can assume that these are initialized correctly and the offset + length is less than the length of the file.

```
off_t offset;
size_t length;
```

We'll assume that all system calls succeed. First, we have to open the file and get the size.

```
struct stat sb;
int fd = open(argv[1], O_RDONLY);
fstat(fd, &sb);
```

Then, we need to introduce another variable known as page_offset. mmap doesn't let the program pass in any value as an offset, it needs to be a multiple of the page size. In our case, we will round down.

```
off_t page_offset = offset & ~(sysconf(_SC_PAGE_SIZE) - 1);
```

Then, we make the call to mmap, here is the order of arguments.

1. NULL, this tells mmap we don't need any particular address to start from
2. length + offset - page_offset, mmaps the "rest" of the file into memory (starting from offset)
3. PROT_READ, we want to read the file
4. MAP_PRIVATE, tell the OS, we don't want to share our mapping
5. fd, object descriptor that we refer to
6. pa_offset, the page aligned offset to start from

```
char * addr = mmap(NULL, length + offset - page_offset,
    PROT_READ,
 MAP_PRIVATE, fd, page_offset);
```

Now, we can interact with the address as if it were a normal buffer. After, we have to unmap the file and close the file descriptor to make sure other system resources are freed.

```
write(1, addr + offset - page_offset, length);
munmap(addr, length + offset - pa_offset);
close(fd);
```

Check out the full listing in the man pages.

MMAP Communication

So how would we use mmap to communicate across processes? Conceptually, it would be the same as using threading. Let's go through a broken down example. First, we need to allocate some space. We can do that with the `mmap` call. We'll also allocate space for 100 integers

```
int size = 100 * sizeof(int);
void *addr = mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED |
    MAP_ANONYMOUS, -1, 0);
int *shared = addr;
```

Then, we need to fork and perform some communication. Our parent will store some values, and our child will read those values.

```
pid_t mychild = fork();
if (mychild > 0) {
 shared[0] = 10;
 shared[1] = 20;
} else {
 sleep(1); // Check the synchronization chapter for a better way
 printf("%d\n", shared[1] + shared[0]);
}
```

Now, there is no assurance that the values will be communicated because the process used `sleep`, not a mutex. Most of the time this will work.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h> /* mmap() is defined in this header */
```

```
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main() {

 int size = 100 * sizeof(int);
 void *addr = mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED
     | MAP_ANONYMOUS, -1, 0);

 printf("Mapped at %p\n", addr);

 int *shared = addr;
 pid_t mychild = fork();
 if (mychild > 0) {
 shared[0] = 10;
 shared[1] = 20;
 } else {
 sleep(1); // We will talk about synchronization later
 printf("%d\n", shared[1] + shared[0]);
 }

 munmap(addr,size);
 return 0;
}
```

This piece of code allocates space for a 100 integers and creates a piece of memory that is shared between all processes. The code then forks. The parent process writes two integers to the first two slots. To avoid a data race, the child sleeps for a second and then prints out the stored values. This is an imperfect way to protect against data races. We could use a mutex across the processes mentioned in the synchronization section. But for this simple example, it works fine. Note that each process should call munmap when done using the piece of memory.

Sharing anonymous memory is an efficient form of inter-process communication because there is no copying, system call, or disk-access overhead - the two processes share the same physical frame of main memory. On the other hand, shared memory, like in a multithreading context, creates room for data races. Processes that share writable memory might need to use synchronization primitives like mutexes to prevent these from happening.

Pipes

You've seen the virtual memory way of IPC, but there are more standard versions of IPC that are provided by the kernel. One of the big utilities is POSIX pipes. A pipe simply takes in a stream of bytes and spits out a sequence of bytes.

One of the big starting points of pipes was way back in the PDP-10 days. In those days, a write to the disk or even your terminal was slow as it may have to be printed out. The Unix programmers still wanted to create small, portable programs that did one thing well and could be composed. As such, pipes were invented to take the output of one program and feed it to the input of another program though they have other uses today – you can read more At the Wikipedia page Consider if you type the following into your terminal.

```
$ ls -1 | cut -d'.' -f1 | sort | uniq | tee dirents
```

What does the following code do? First, it lists the current directory. The -1 means that it outputs one entry per line. The `cut` command then takes everything before the first period. `sort` sorts all the input lines, `uniq` makes sure all the lines are unique. Finally, `tee` outputs the contents to the file `dir_contents` and the terminal for your perusal. The important part is that bash creates **5 separate processes** and connects their standard outs/stdins with pipes the trail looks something like this.

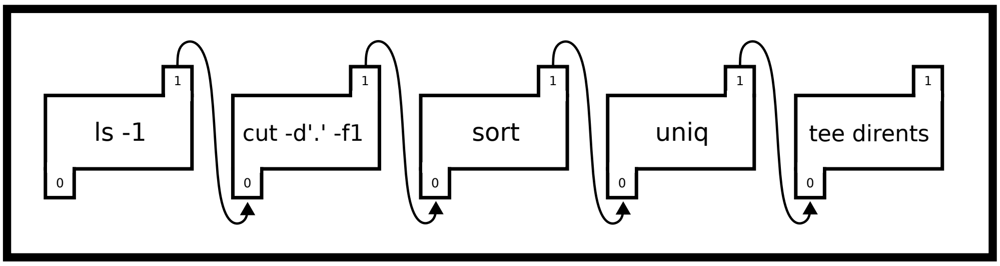


Figure 9.8: Pipe Process Filedescriptor redirection

The numbers in the pipes are the file descriptors for each process and the arrow represents the redirect or where the output of the pipe is going. A POSIX pipe is almost like its real counterpart - a program can stuff bytes down one end and they will appear at the other end in the same order. Unlike real pipes, however, the flow is always in the same direction, one file descriptor is used for reading and the other for writing. The `pipe` system call is used to create a pipe. These file descriptors can be used with `read` and `write`. A common method of using pipes is to create the pipe before forking to communicate with a child process

```
int filedes[2];
pipe (filedes);
pid_t child = fork();
if (child > 0) {/* I must be the parent */
 char buffer[80];
 int bytesread = read(filedes[0], buffer, sizeof(buffer));
 // do something with the bytes read
} else {
 write(filedes[1], "done", 4);
}
```

There are two file descriptors that pipe creates. `filedes[0]` contains the read end. `filedes[1]` contains the write end. How your friendly neighborhood TAs remember it is one can *read before they can write, or reading comes before writing*. You can groan all you want at it, but it is helpful to remember what is the read end and what is the write end.

One can use pipes inside of the same process, but there tends to be no added benefit. Here's an example program that sends a message to itself.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
 int fh[2];
 pipe(fh);
 FILE *reader = fdopen(fh[0], "r");
 FILE *writer = fdopen(fh[1], "w");
 // Hurrah now I can use printf
 printf("Writing...\n");
 fprintf(writer,"%d %d %d\n", 10, 20, 30);
 fflush(writer);

 printf("Reading...\n");
 int results[3];
 int ok = fscanf(reader,"%d %d %d", results, results + 1,
     results + 2);
 printf("%d values parsed: %d %d %d\n", ok, results[0],
     results[1], results[2]);

 return 0;
}
```

The problem with using a pipe in this fashion is that writing to a pipe can block meaning the pipe only has a limited buffering capacity. The maximum size of the buffer is system-dependent; typical values from 4KiB up to 128KiB though they can be changed.

```
int main() {
 int fh[2];
 pipe(fh);
 int b = 0;
 #define MESG "..............................."
 while(1) {
 printf("%d\n",b);
 write(fh[1], MESG, sizeof(MESG))
 b+=sizeof(MESG);
 }
 return 0;
}
```

Pipe Gotchas

Here's a complete example that doesn't work! The child reads one byte at a time from the pipe and prints it out - but we never see the message! Can you see why?

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main() {
 int fd[2];
 pipe(fd);
 //You must read from fd[0] and write from fd[1]
 printf("Reading from %d, writing to %d\n", fd[0], fd[1]);

 pid_t p = fork();
 if (p > 0) {
 /* I have a child, therefore I am the parent */
 write(fd[1],"Hi Child!",9);

 /*don't forget your child*/
 wait(NULL);
 } else {
 char buf;
 int bytesread;
 // read one byte at a time.
 while ((bytesread = read(fd[0], &buf, 1)) > 0) {
 putchar(buf);
 }
 }
 return 0;
}
```

The parent sends the bytes `H,i,(space),C...!` into the pipe. The child starts reading the pipe one byte at a time. In the above case, the child process will read and print each character. However, it never leaves the while loop! When there are no characters left to read it simply blocks and waits for more unless **All the writers close their ends** Another solution could also exit the loop by checking for an end-of-message marker,

```
while ((bytesread = read(fd[0], &buf, 1)) > 0) {
 putchar(buf);
 if (buf == '!') break; /* End of message */
}
```

We know that when a process tries to read from a pipe where there are still writers, the process blocks. If no pipe has no writers, read returns 0. If a process tries to write with some reader's read goes through, or fails – partially or completely – if the pipe is full. Why happens when a process tries to write when there are no readers left?

```
If all file descriptors referring to the read end of a pipe have been closed,
then a write(2) will cause a SIGPIPE signal to be generated for the calling process.
```

Tip: Notice only the writer (not a reader) can use this signal. To inform the reader that a writer is closing their end of the pipe, a program could write your special byte (e.g. 0xff) or a message (`"Bye!"`)

Here's an example of catching this signal that fails! Can you see why?

```c
#include <stdio.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void no_one_listening(int signal) {
 write(1, "No one is listening!\n", 21);
}

int main() {
 signal(SIGPIPE, no_one_listening);
 int filedes[2];

 pipe(filedes);
 pid_t child = fork();
 if (child > 0) {
 /* This process is the parent. Close the listening end of the
     pipe */
 close(filedes[0]);
 } else {
 /* Child writes messages to the pipe */
 write(filedes[1], "One", 3);
 sleep(2);
 // Will this write generate SIGPIPE ?
 write(filedes[1], "Two", 3);
 write(1, "Done\n", 5);
 }
 return 0;
}
```

The mistake in the above code is that there is still a reader for the pipe! The child still has the pipe's first file descriptor open and remember the specification? All readers must be closed

When forking, *It is common practice* to close the unnecessary (unused) end of each pipe in the child and parent process. For example, the parent might close the reading end and the child might close the writing end.

The last addendum is that a program can set the file descriptor to return when there is no one listening instead of SIGPIPE because by default SIGPIPE terminates your program. The reason that this is default behavior is it makes the pipe example above work. Consider this useless use of cat

```
$ cat /dev/urandom | head -n 20
```

Which grabs 20 lines of input from urandom. `head` will terminate after 20 newline characters have been read. What about `cat`? `cat` needs to receive a SIGPIPE informing it that the program tried to write to a pipe that no one is listening on.

Other pipe facts

A pipe gets filled up when the writer writes too much to the pipe without the reader reading any of it. When the pipes become full, all writes fail until a read occurs. Even then, a write may partially fail if the pipe has a little bit of space left but not enough for the entire message. Usually, two things are done to avoid this. Either increase the size of the pipe. Or more commonly, fix your program design so that the pipe is constantly being read from.

As hinted at before, Pipe writes are atomic up to the size of the pipe. Meaning that if two processes try to write to the same pipe, the kernel has internal mutexes with the pipe that it will lock, do the write, and return. The only gotcha is when the pipe is about to become full. If two processes are trying to write and the pipe can only satisfy a partial write, that pipe write is not atomic – be careful about that!

Unnamed pipes live in memory and are a simple and efficient form of inter-process communication (IPC) that is useful for streaming data and simple messages. Once all processes have closed, the pipe resources are freed.

It is also common design for a pipe to be one way – meaning one process should do the writing and one process do the reading. Otherwise, the child would attempt to read its data intended for the parent (and vice versa)!

Pipes and Dup

Often, you'll want to use `pipe2` in combination with dup. Take for example the simple program in the command line.

```
$ ls -1 | cut -f1 -d.
```

This command takes the output of `ls -1` which lists the content of the current directory on one line each and pipes it to cut. Cut take a delimiter, in this case, a dot, and a field position, in our case 1, and outputs per line the nth field by each delimiter. At a high level, this grabs the file names without the extension of our current directory.

Underneath the hood, this is how bash does it internally.

```
#define _GNU_SOURCE

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

```
#include <unistd.h>
#include <stdlib.h>

int main() {

 int pipe_fds[2];
 // Call with the O_CLOEXEC flag to prevent any commands from
     blocking
 pipe2(pipe_fds, O_CLOEXEC);

 // Remember for pipe_fds, the program read then write (reading
     is 0 and writing is 1)

 if(!fork()) {
 // Child

 // Make the stdout of the process, the write end
 dup2(pipe_fds[1], 1);

 // Exec! Don't forget the cast
 execlp("ls", "ls", "-1", (char*)NULL);
 exit(-1);
 }

 // Same here, except the stdin of the process is the read end
 dup2(pipe_fds[0], 0);

 // Same deal here
 execlp("cut", "cut", "-f1", "-d.", (char*)NULL);
 exit(-1);

 return 0;
}
```

The results of the two programs should be the same. Remember as you encounter more complicated examples of piping processes up, a program needs to close all unused ends of pipes otherwise the program will deadlock waiting for your processes to finish.

Pipe Conveniences

If the program already has a file descriptor, it can 'wrap' it yourself into a FILE pointer using fdopen.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int main() {
 char *name="Fred";
 int score = 123;
 int filedes = open("mydata.txt", "w", O_CREAT, S_IWUSR |
     S_IRUSR);

 FILE *f = fdopen(filedes, "w");
 fprintf(f, "Name:%s Score:%d\n", name, score);
 fclose(f);
```

For writing to files, this is unnecessary. Use fopen which does the same as open and fdopen. However for pipes, we already have a file descriptor, so this is a great time to use fdopen

Here's a complete example using pipes that almost works! Can you spot the error? Hint: The parent never prints anything!

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
 int fh[2];
 pipe(fh);
 FILE *reader = fdopen(fh[0], "r");
 FILE *writer = fdopen(fh[1], "w");
 pid_t p = fork();
 if (p > 0) {
 int score;
 fscanf(reader, "Score %d", &score);
 printf("The child says the score is %d\n", score);
 } else {
 fprintf(writer, "Score %d", 10 + 10);
 fflush(writer);
 }
 return 0;
}
```

Note the unnamed pipe resource will disappear once both the child and parent have exited. In the above example, the child will send the bytes and the parent will receive the bytes from the pipe. However, no end-of-line character is ever sent, so fscanf will continue to ask for bytes because it is waiting for the end of the line i.e. it will wait forever! The fix is to ensure we send a newline character so that fscanf will return.

```
change: fprintf(writer, "Score %d", 10 + 10);
to: fprintf(writer, "Score %d\n", 10 + 10);
```

If you want your bytes to be sent to the pipe immediately, you'll need to fflush! Remember back to the introduction section that shows the difference between terminal vs non-terminal outputs of stdout.

Even though we have a section on it, it is highly not recommended to use the file descriptor API for non-seekable files. The reason being is that while we get conveniences we also get annoyances like the buffering example we mentioned, caching, etc. The basic C library motto is that any device a program can properly `fseek` or move to an arbitrary position, it should be able to `fdopen`. Files satisfy this behavior, shared memory also, terminals, etc. When it comes to pipes, sockets, epoll objects, etc, don't do it.

Named Pipes

An alternative to *unnamed* pipes is *named* pipes created using `mkfifo`. From the command line: `mkfifo` From C: `int mkfifo(const char *pathname, mode_t mode);`

You give it the pathname and the operation mode, it will be ready to go! Named pipes take up virtually no space on a file system. This means the actual contents of the pipe aren't printed to the file and read from that same file. What the operating system tells you when you have a named pipe is that it will create an unnamed pipe that refers to the named pipe, and that's it! There is no additional magic. This is for programming convenience if processes are started without forking meaning that there would be no way to get the file descriptor to the child process for an unnamed pipe.

Hanging Named Pipes

A named pipe `mkfifo` is a pipe that a program calls `open(2)` on with read and/or write permissions. This is useful if you want to have a pipe between two processes without one processing having to fork the other process. There are some gotchas with named pipes. There is more down below, but we'll introduce it here for a simple example. Reads and writes hang on Named Pipes until there is at least one reader and one writer, take this.

```
1$ mkfifo fifo
1$ echo Hello > fifo
# This will hang until the following command is run on another
   terminal or another process
2$ cat fifo
Hello
```

Any `open` is called on a named pipe the kernel blocks until another process calls the opposite open. Meaning, echo calls `open(.., O_RDONLY)` but that blocks until cat calls `open(.., O_WRONLY)`, then the programs are allowed to continue.

Race condition with named pipes

What is wrong with the following program?

```
//Program 1

int main(){
 int fd = open("fifo", O_RDWR | O_TRUNC);
 write(fd, "Hello!", 6);
 close(fd);
 return 0;
}

//Program 2
int main() {
 char buffer[7];
 int fd = open("fifo", O_RDONLY);
 read(fd, buffer, 6);
 buffer[6] = '\0';
 printf("%s\n", buffer);
 return 0;
}
```

This may never print hello because of a race condition. Since a program opened the pipe in the first process under both permissions, open won't wait for a reader because the program told the operating system that it is a reader! Sometimes it looks like it works because the execution of the code looks something like this.

Table 9.1: Fine Pipe Access Pattern

	Process 1	Process 2
Time 1	open(O_RDWR) & write()	
Time 2		open(O_RDONLY) & read()
Time 3	close() & exit()	
Time 4	print() & exit()	

But here is an invalid series of operations that cause a race condition.

Table 9.2: Pipe Race Condition

	Process 1	Process 2
Time 1	open(O_RDWR) & write()	
Time 2	close() & exit()	
Time 3		open(O_RDONLY) (Blocks Indefinitely)

Files

On Linux, there are two abstractions with files. The first is the Linux `fd` level abstraction.

- `open` takes a path to a file and creates a file descriptor entry in the process table. If the file is inaccessible, it errors out.
- `read` takes a certain number of bytes that the kernel has received and reads them into a user-space buffer. If the file is not open in read mode, this will break.
- `write` outputs a certain number of bytes to a file descriptor. If the file is not open in write mode, this will break. This may be buffered internally.
- `close` removes a file descriptor from a process' file descriptors. This always succeeds for a valid file descriptor.
- `lseek` takes a file descriptor and moves it to a certain position. It can fail if the seek is out of bounds.
- `fcntl` is the catch-all function for file descriptors. Set file locks, read, write, edit permissions, etc.

The Linux interface is powerful and expressive, but sometimes we need portability for example if we are writing for a Macintosh or windows. This is where C's abstraction comes into play. On different operating systems, C uses the low-level functions to create a wrapper around files used everywhere, meaning that C on Linux uses the above calls.

- `fopen` opens a file and returns an object. `null` is returned if the program doesn't have permission for the file.
- `fread` reads a certain number of bytes from a file. An error is returned if already at the end of the file when which the program must call `feof()` to check if the program attempted to read *past* the end of the file.
- `fgetc/fgets` Get a char or a string from a file
- `fscanf` Read a format string from the file
- `fwrite` Write some objects to a file
- `fprintf` Write a formatted string to a file
- `fclose` Close a file handle
- `fflush` Take any buffered changes and flush them to a file

But programs don't get the expressiveness that Linux gives with system calls. A program can convert back and forth between them with `int fileno(FILE* stream)` and `FILE* fdopen(int fd...)`. Also, C files are **buffered** meaning that their contents may be written to the backing after the call returns. You can change that with C options.

Danger With portability you lose something important, the ability to tell an error. A program can `fopen` a file descriptor and get a `FILE*` object but **it won't be the same as a file** meaning that certain calls will fail or act weirdly. The C API reduces this weirdness, but for example a program cannot `fseek` to a part of the file, or perform any operations with its buffering. The problem is the API won't give a lot of warning because C needs to maintain compatibility with other operating systems. To keep things simple, use the C API of files when dealing with a file on disk, which will work fine. Otherwise, be in for a rough ride for portability's sake.

Determining File Length

For files less than the size of a long, using fseek and ftell is a simple way to accomplish this. Move to the end of the file and find out the current position.

```
fseek(f, 0, SEEK_END);
long pos = ftell(f);
```

This tells us the current position in the file in bytes - i.e. the length of the file!
`fseek` can also be used to set the absolute position.

```
fseek(f, 0, SEEK_SET); // Move to the start of the file
fseek(f, posn, SEEK_SET); // Move to 'posn' in the file.
```

All future reads and writes in the parent or child processes will honor this position. Note writing or reading from the file will change the current position. See the man pages for fseek and ftell for more information.

Use stat instead

This only works on some architectures and compilers That quirk is that longs only need to be **4 Bytes big** meaning that the maximum size that `ftell` can return is a little under 2 Gibibytes. Nowadays, our files could be hundreds of gibibytes or even terabytes on a distributed file system. What should we do instead? Use `stat`! We will cover stat in a later part but here is some code that will tell a program the size of the file

```
struct stat buf;
if(stat(filename, &buf) == -1){
 return -1;
}
return (ssize_t)buf.st_size;
```

`buf.st_size` is of type `off_t` which is big enough for large files.

Gotchas with files

What happens when file streams are closed by two different processes? Closing a file stream is unique to each process. Other processes can continue to use their file handles. Remember, everything is copied over when a child is created, even the relative positions of the files. As you might have observed with using `fork`, there is a quirk of the implementation of files and their caches on Ubuntu that will rewind a file descriptor once a file has been closed. As such, make sure to close before forking or at least don't trigger a cache inconsistency which is much harder.

IPC Alternatives

Okay so now you have a list of tools in your toolbox to tackle communicating between processes, so what should you use?

There is no hard answer, though this is the most interesting question. Generally, we have retained pipes for legacy reasons. This means that we only use them to redirect stdin, stdout, and stderr for the collection of logs and similar programs. You may find processes trying to communicate with unnamed or named pipes as well. Most of the time you won't be dealing with this interaction directly though.

Files are used almost all the time as a form of IPC. Hadoop is a great example where processes will write to append-only tables and then other processes will read from those tables. We generally use files under a few cases. One case is if we want to save the intermediate results of an operation to a file for future use. Another case is if putting it in memory would cause an out of memory error. On Linux, file operations are generally pretty cheap, so most programmers use it for larger intermediate storage.

mmap is used for two scenarios. One is a `linear` or `near-linear` read through of the file. Meaning, a program reads the file front to back or back to front. The key is that the program doesn't jump around too much. Jumping around too much causes thrashing and loses all the benefits of using mmap. The other usage of mmap is for direct memory inter-process communication. This means that a program can store structures in a piece of mmap'ed memory and share them between two processes. Python and Ruby use this mapping all the time to utilize copy on write semantics.

Topics

1. Virtual Memory
2. Page Table
3. MMU/TLB
4. Address Translation
5. Page Faults
6. Frames/Pages
7. Single-level vs multi-level page table
8. Calculating offsets for multi-level page table
9. Pipes
10. Pipe read write ends
11. Writing to a zero reader pipe
12. Reading from a zero writer pipe
13. Named pipe and Unnamed Pipes
14. Buffer Size/Atomicity
15. Scheduling Algorithms
16. Measures of Efficiency

Questions

1. What is virtual memory?
2. What are the following and what is their purpose?
   (a) Translation Lookaside Buffer
   (b) Physical Address
   (c) Memory Management Unit. Multilevel page table. Frame number. Page number and page offset.
   (d) The dirty bit
   (e) The NX Bit
3. What is a page table? How about a physical frame? Does a page always need to point to a physical frame?
4. What is a page fault? What are the types? When does it result in a SEGFAULT?
5. What are the advantages to a single-level page table? Disadvantages? How about a multi-level table?
6. What does a multi-leveled table look like in memory?
7. How do you determine how many bits are used in the page offset?
8. Given a 64-bit address space, 4kb pages and frames, and a 3 level page table, how many bits are the Virtual page number 1, VPN2, VPN3 and the offset?
9. What is a pipe? How do we create pipes?
10. When is SIGPIPE delivered to a process?
11. Under what conditions will calling read() on a pipe block? Under what conditions will read() immediately return 0
12. What is the difference between a named pipe and an unnamed pipe?
13. Is a pipe thread-safe?
14. Write a function that uses fseek and ftell to replace the middle character of a file with an 'X'
15. Write a function that creates a pipe and uses write to send 5 bytes, "HELLO" to the pipe. Return the read file descriptor of the pipe.
16. What happens when you mmap a file?
17. Why is getting the file size with ftell not recommended? How should you do it instead?

Bibliography

[1] Dec pdp-10 ka10 control panel. URL `http://www.ricomputermuseum.org/Home/interesting_computer_items/dec-pdp-ka10`.

[2] mmap, Jul 2018. URL `http://pubs.opengroup.org/onlinepubs/9699919799/functions/mmap.html`.

[3] International Business Machines Corporation (IBM). *IBM 709 Data Processing System Reference Manual*. International Business Machines Corporation (IBM). URL http://archive.computerhistory.org/resources/text/Fortran/102653991.05.01.acc.pdf.

10 Scheduling

I wish that I could fly
There's danger if I dare to stop and here's the reason why
You see I'm overdue
I'm in a rabbit stew
Can't even say "Good-bye", hello
I'm late, I'm late, I'm late
No, no, no, no, no, no, no!

Alice in Wonderland

CPU Scheduling is the problem of efficiently selecting which process to run on a system's CPU cores. In a busy system, there will be more ready-to-run processes than there are CPU cores, so the system kernel must evaluate which processes should be scheduled to run and which processes should be executed later. The system must also decide whetherit should take a particular process and pause its execution – along with any associated threads. The balance comes from stopping processes often enough where you have a responsive computer but infrequently enough where the programs themselves are spending minimal time context switching. It is a hard balance to get right.

The additional complexity of multi-threaded and multiple CPU cores are considered a distraction to this initial exposition so are ignored here. Another gotcha for non-native speakers is the dual meaning of "Time": The word "Time" can be used in both clock and elapsed duration context. For example "The arrival time of the first process was 9:00am." and, "The running time of the algorithm is 3 seconds".

One clarification that we will make is that our scheduling will mainly deal with short term or CPU scheduling. That means we will assume that the processes are in memory and ready to go. The other types of scheduling are long and medium term. Long term schedulers act as gatekeepers to the processing world. When a process requests another process to be executed, it can either tell the process yes, no, or wait. The medium term scheduler deals with the caveats of moving a process from the paused state in memory to the paused state on disk when there are too many processes or some process are known to use an insignificant amount of CPU cycles. Think about a process that only checks something once an hour.

High Level Scheduler Overview

Schedulers are pieces of software programs. In fact, you can implement schedulers yourself! If you are given a list of commands to exec, a program can schedule them them with SIGSTOP and SIGCONT. These are called user space schedulers. Hadoop and python's celery may do some sort of user space scheduling or deal with the operating system.

At the operating system level, you generally have this type of flowchart, described in words first below. Note, please don't memorize all the states.

1. New is the initial state. A process has been requested to schedule. All process requests come from fork or clone. At this point the operating system knows it needs to create a new process.

2. A process moves from the new state to the ready. This means any structs in the kernel are allocated. From there, it can go into ready suspended or running.

3. Running is the state that we hope most of our processes are in, meaning they are doing useful work. A process could either get preempted, blocked, or terminate. Preemption brings the process back to the ready state. If a process is blocked, that means it could be waiting on a mutex lock, or it could've called sleep – either way, it willingly gave up control.

4. On the blocked state the operating system can either turn the process ready or it can go into a deeper state called blocked suspended.

5. There are so-called deep slumber states called blocked suspended and blocked ready. You don't need to worry about these.

We will try to pick a scheme that decides when a process should move to the running state, and when it should be moved back to the ready state. We won't make much mention of how to factor in voluntarily blocked states and when to switch to deep slumber states.

Measurements

Scheduling affects the performance of the system, specifically the *latency* and *throughput* of the system. The throughput might be measured by a system value, for example, the I/O throughput - the number of bits written per second, or the number of small processes that can complete per unit time. The latency might be measured by the response time – elapse time before a process can start to send a response – or wait time or turnaround time –the elapsed time to complete a task. Different schedulers offer different optimization trade-offs that may be appropriate for desired use. There is no optimal scheduler for all possible environments and goals. For example, Shortest Job First will minimize total wait time across all jobs but in interactive (UI) environments it would be preferable to minimize response time at the expense of some throughput, while FCFS seems intuitively fair and easy to implement but suffers from the Convoy Effect. Arrival time is the time at which a process first arrives at the ready queue, and is ready to start executing. If a CPU is idle, the arrival time would also be the starting time of execution.

What is preemption?

Without preemption, processes will run until they are unable to utilize the CPU any further. For example the following conditions would remove a process from the CPU and the CPU would be available to be scheduled for other processes. The process terminates due to a signal, is blocked waiting for concurrency primitive, or exits normally. Thus once a process is scheduled it will continue even if another process with a high priority appears on the ready queue.

With preemption, the existing processes may be removed immediately if a more preferred process is added to the ready queue. For example, suppose at t=0 with a Shortest Job First scheduler there are two processes (P1 P2) with 10 and 20 ms execution times. P1 is scheduled. P1 immediately creates a new process P3, with execution time of 5 ms, which is added to the ready queue. Without preemption, P3 will run 10ms later (after P1 has completed). With preemption, P1 will be immediately evicted from the CPU and instead placed back in the ready queue, and P3 will be executed instead by the CPU.

Any scheduler that doesn't use some form of preemption can result in starvation because earlier processes may never be scheduled to run (assigned a CPU). For example with SJF, longer jobs may never be scheduled if the system continues to have many short jobs to schedule. It all depends on the type of scheduler.

Why might a process (or thread) be placed on the ready queue?

A process is placed on the ready queue when it can use a CPU. Some examples include:

- A process was blocked waiting for a `read` from storage or socket to complete and data is now available.
- A new process has been created and is ready to start.
- A process thread was blocked on a synchronization primitive (condition variable, semaphore, mutex lock) but is now able to continue.
- A process is blocked waiting for a system call to complete but a signal has been delivered and the signal handler needs to run.

Measures of Efficiency

First some definitions

1. `start_time` is the wall-clock start time of the process (CPU starts working on it)
2. `end_time` is the end wall-clock of the process (CPU finishes the process)
3. `run_time` is the total amount of CPU time required
4. `arrival_time` is the time the process enters the scheduler (CPU may start working on it)

Here are measures of efficiency and their mathematical equations

1. `Turnaround Time` is the total time from when the process arrives to when it ends. `end_time - arrival_time`
2. `Response Time` is the total latency (time) that it takes from when the process arrives to when the CPU actually starts working on it. `start_time - arrival_time`

3. `Wait Time` is the *total* wait time or the total time that a process is on the ready queue. A common mistake is to believe it is only the initial waiting time in the ready queue. If a CPU intensive process with no I/O takes 7 minutes of CPU time to complete but required 9 minutes of wall-clock time to complete we can conclude that it was placed on the ready-queue for 2 minutes. For those 2 minutes, the process was ready to run but had no CPU assigned. It does not matter when the job was waiting, the wait time is 2 minutes. `end_time - arrival_time - run_time`

Convoy Effect

The convoy effect is when a process takes up a lot of the CPU time, leaving all other processes with potentially smaller resource needs following like a Convoy Behind them.

Suppose the CPU is currently assigned to a CPU intensive task and there is a set of I/O intensive processes that are in the ready queue. These processes require a tiny amount of CPU time but they are unable to proceed because they are waiting for the CPU-intensive task to be removed from the processor. These processes are starved until the CPU bound process releases the CPU. But, the CPU will rarely be released. For example, in the case of an FCFS scheduler, we must wait until the process is blocked due to an I/O request. The I/O intensive process can now finally satisfy their CPU needs, which they can do quickly because their CPU needs are small and the CPU is assigned back to the CPU-intensive process again. Thus the I/O performance of the whole system suffers through an indirect effect of starvation of CPU needs of all processes.

This effect is usually discussed in the context of FCFS scheduler; however, a Round Robin scheduler can also exhibit the Convoy Effect for long time-quanta.

Extra: Linux Scheduling

As of February 2016, Linux by default uses the *Completely Fair Scheduler* for CPU scheduling and the Budget Fair Scheduling “BFQ” for I/O scheduling. Appropriate scheduling can have a significant impact on throughput and latency. Latency is important for interactive and soft-real time applications such as audio and video streaming. See the discussion and comparative benchmarks here for more information.

Here is how the CFS schedules

- The CPU creates a Red-Black tree with the processes virtual runtime (runtime / nice_value) and sleeper fairness flag – if the process is waiting on something, give it the CPU when it is done waiting.

- Nice values are the kernel’s way of giving priority to certain processes, the lower nice value the higher priority.

- The kernel chooses the lowest one based on this metric and schedules that process to run next, taking it off the queue. Since the red-black tree is self-balancing this operation is guaranteed $O(log(n))$ (selecting the min process is the same runtime)

Although it is called the Fair Scheduler there are a fair bit of problems.

- Groups of processes that are scheduled may have imbalanced loads so the scheduler roughly distributes the load. When another CPU gets free it can only look at the average load of a group schedule, not the individual cores. So the free CPU may not take the work from a CPU that is burning so long as the average is fine.

- If a group of processes is running on non-adjacent cores then there is a bug. If the two cores are more than a hop away, the load balancing algorithm won't even consider that core. Meaning if a CPU is free and a CPU that is doing more work is more than a hop away, it won't take the work (may have been patched).
- After a thread goes to sleep on a subset of cores, when it wakes up it can only be scheduled on the cores that it was sleeping on. If those cores are now busy, the thread will have to wait on them, wasting opportunities to use other idle cores.
- To read more on the problems of the Fair Scheduler, read here.

Scheduling Algorithms

Unless otherwise stated

1. Process 1: Runtime 1000ms
2. Process 2: Runtime 2000ms
3. Process 3: Runtime 3000ms
4. Process 4: Runtime 4000ms
5. Process 5: Runtime 5000ms

Shortest Job First (SJF)

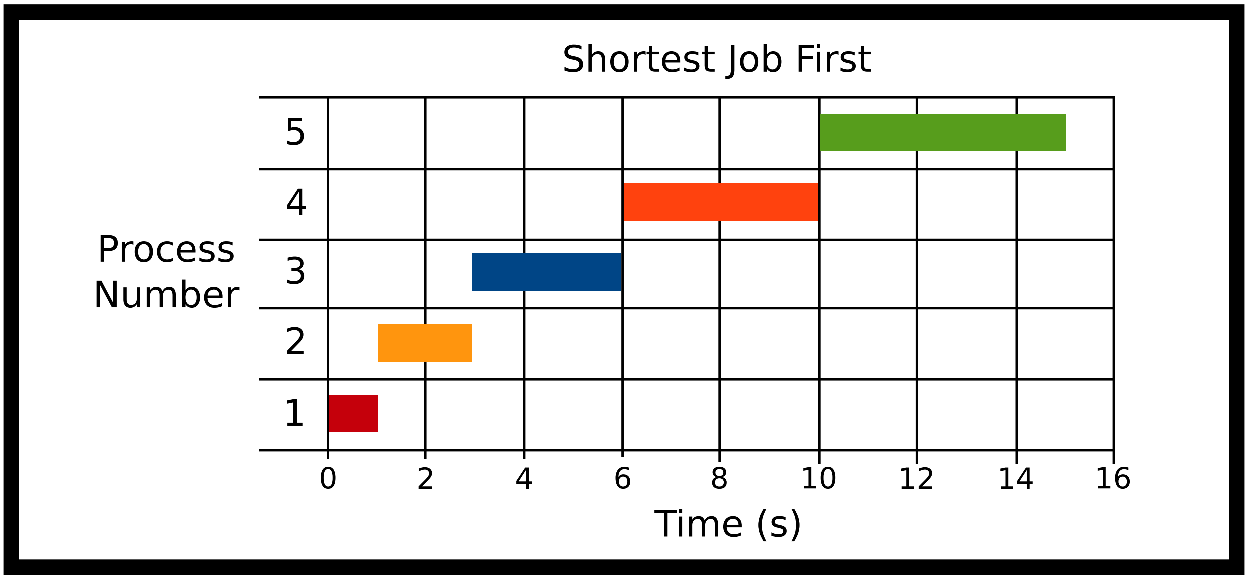


Figure 10.1: Shortest job first scheduling

- P1 Arrival: 0ms

- P2 Arrival: 0ms
- P3 Arrival: 0ms
- P4 Arrival: 0ms
- P5 Arrival: 0ms

The processes all arrive at the start and the scheduler schedules the job with the shortest total CPU time. The glaring problem is that this scheduler needs to know how long this program will run over time before it ran the program.

Technical Note: A realistic SJF implementation would not use the total execution time of the process but the burst time or the number of CPU cycles needed to finish a program. The expected burst time can be estimated by using an exponentially decaying weighted rolling average based on the previous burst time [3, Chapter 6]. For this exposition, we will simplify this discussion to use the total running time of the process as a proxy for the burst time.

Advantages

1. Shorter jobs tend to get run first
2. On average wait times and response times are down

Disadvantages

1. Needs algorithm to be omniscient
2. Need to estimate the burstiness of a process which is harder than let's say a computer network

Preemptive Shortest Job First (PSJF)

Preemptive shortest job first is like shortest job first but if a new job comes in with a shorter runtime than the total runtime of the current job, it is run instead. If it is equal like our example our algorithm can choose. The scheduler uses the *total* runtime of the process. If the scheduler wants to compare the shortest *remaining* time left, that is a variant of PSJF called Shortest Remaining Time First (SRTF).

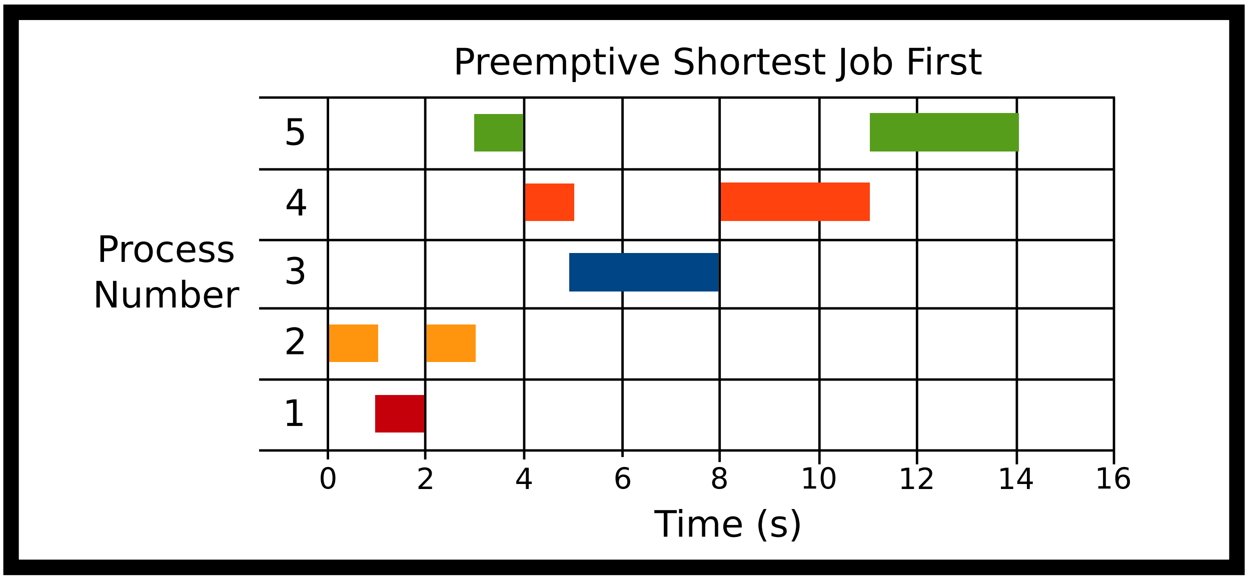


Figure 10.2: Preemptive Shortest Job First scheduling

- P2 at 0ms
- P1 at 1000ms
- P5 at 3000ms
- P4 at 4000ms
- P3 at 5000ms

Here's what our algorithm does. It runs P2 because it is the only thing to run. Then P1 comes in at 1000ms, P2 runs for 2000ms, so our scheduler preemptively stops P2, and let's P1 run all the way through. This is completely up to the algorithm because the times are equal. Then, P5 Comes in – since no processes running, the scheduler will run process 5. P4 comes in, and since the runtimes are equal P5, the scheduler stops P5 and runs P4. Finally, P3 comes in, preempts P4, and runs to completion. Then P4 runs, then P5 runs.

Advantages

1. Ensures shorter jobs get run first

Disadvantages

1. Need to know the runtime again
2. Context switching and jobs can get interrupted

First Come First Served (FCFS)

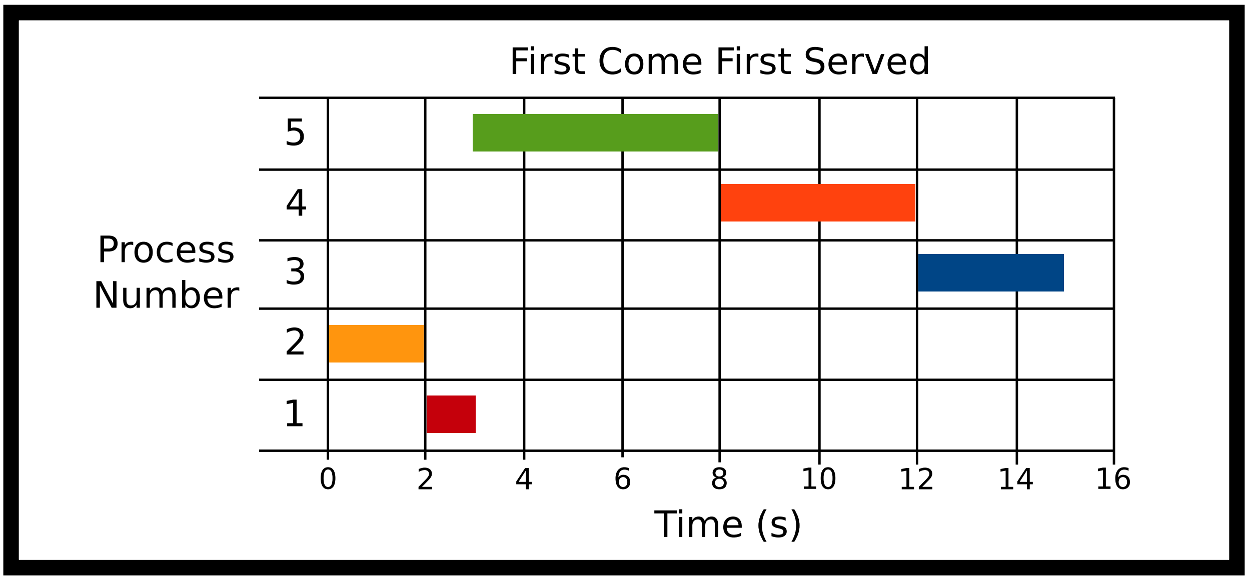


Figure 10.3: First come first serve scheduling

- P2 at 0ms
- P1 at 1000ms
- P5 at 3000ms
- P4 at 4000ms
- P3 at 5000ms

Processes are scheduled in the order of arrival. One advantage of FCFS is that scheduling algorithm is simple The ready queue is a FIFO (first in first out) queue. FCFS suffers from the Convoy effect. Here P2 Arrives, then P1 arrives, then P5, then P4, then P3. You can see the convoy effect for P5.

Advantages

- Simple algorithm and implementation
- Context switches infrequent when there are long-running processes
- No starvation if all processes are guaranteed to terminate

Disadvantages

- Simple algorithm and implementation
- Context switches infrequent when there are long-running processes

Round Robin (RR)

Processes are scheduled in order of their arrival in the ready queue. After a small time step though, a running process will be forcibly removed from the running state and placed back on the ready queue. This ensures long-running processes refrain from starving all other processes from running. The maximum amount of time that a process can execute before being returned to the ready queue is called the time quanta. As the time quanta approaches to infinity, Round Robin will be equivalent to FCFS.

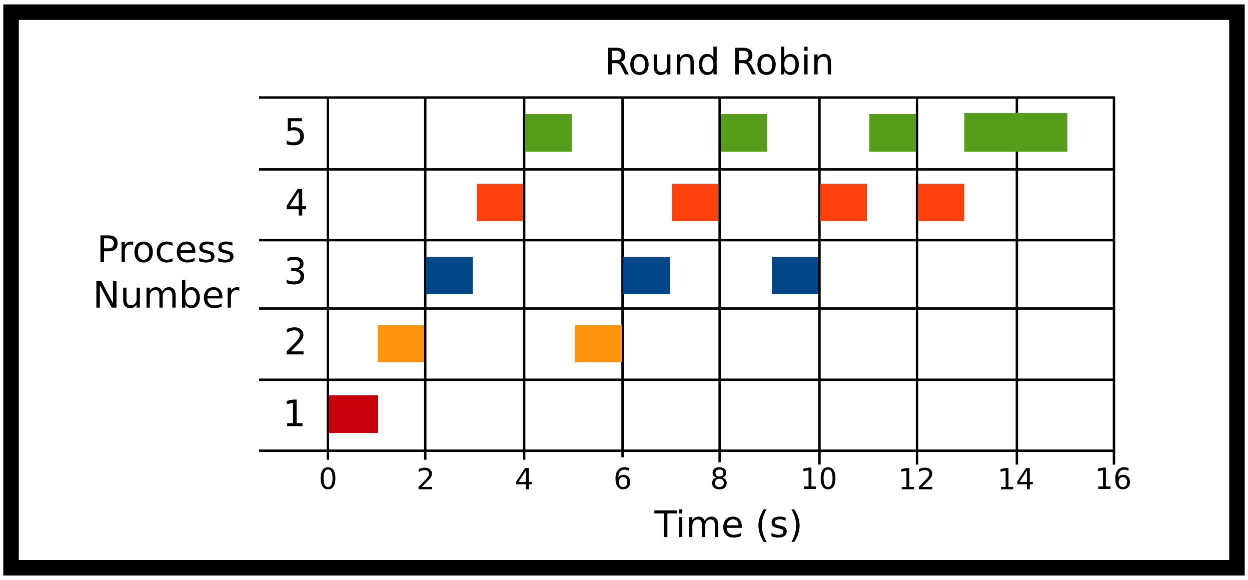


Figure 10.4: Round Robin Scheduling

- P1 Arrival: 0ms
- P2 Arrival: 0ms
- P3 Arrival: 0ms
- P4 Arrival: 0ms
- P5 Arrival: 0ms

Quantum = 1000ms

Here all processes arrive at the same time. P1 is run for 1 quantum and is finished. P2 for one quantum; then, it is stopped for P3. After all other processes run for a quantum we cycle back to P2 until all the processes are finished.

Advantages

1. Ensures some notion of fairness

Disadvantages

1. Large number of processes = Lots of switching

Priority

Processes are scheduled in the order of priority value. For example, a navigation process might be more important to execute than a logging process.

Extra: Scheduling Conceptually

This section could be useful for those that like to analyze these algorithms mathematically

If your co-worker asked you what scheduling algorithm to use, you may not have the tools to analyze each algorithm. So, let's think about scheduling algorithms at a high level and break them down by their times. We will be evaluating this in the context of a random process timing, meaning that each process takes a random but finite amount of time to finish.

Just a refresher, here are the terms.

Table 10.1: Scheduling Variables

Concept	Meaning
Start time	The time the scheduler first started work
End time	When the scheduler finished the process
Arrival time	When the job first arrived at the scheduler
Run time	How long does the process take to run if there is no preemption

And here are the measures we are trying to optimize.

Table 10.2: Scheduling Measures of Efficiency

Measure	Formula
Response Time	Start time minus Arrival time
Turnaround time	End time minus Arrival time
Wait time	End time minus Arrival time minus Run time

Different use cases will be discussed after. Let the maximum amount of time that a process run be equal to S. We will also assume that there are a finite number of processes running at any given time c. Here are some concepts from queueing theory that you'll need to know that will help simplify the theories.

1. Queueing theory involves a random variable controlling the interarrival time – or the time between two different processes arriving. We won't name this random variable, but we will assume that (1) it has a mean of λ and (2) it is distributed as a Poisson random variable. This means the probability of getting a process t units after getting another process is $\lambda^t * \frac{\exp(-\lambda)}{t!}$ where $t!$ can be approximated by the gamma function when dealing with real values.

2. We will be denoting the service time S, and deriving the waiting time W, and the response time R; more specifically the expected values of all of those variables $E[S]$ deriving turnaround time is simply $S + W$. For clarity, we will introduce another variable N that is the number of people currently in the queue. A famous result in queueing theory is Little's Law which states $E[N] = \lambda E[W]$ meaning that the number of people waiting is the arrival rate times the expected waiting time (assuming the queue is in a steady state).

3. We won't make many assumptions about how much time it takes to run each process except that it will take a finite amount of time – otherwise this gets almost impossible to evaluate. We will denote two variables that $\frac{1}{\mu}$ is the mean of the waiting time and that the coefficient of variation C is defined as $C^2 = \frac{var(S)}{E[S]^2}$ to help us control for processes that take a while to finish. An important note is that when $C > 1$ we say that the running times of the process are variadic. We will note below that this rockets up the wait and response times for FCFS quadratically.
4. $\rho = \frac{\lambda}{\mu} < 1$ Otherwise, our queue would become infinitely long
5. We will assume that there is one processor. This is known as an M/G/1 queue in queueing theory.
6. We'll leave the service time as an expectation S otherwise we may run into over-simplifications with the algebra. Plus it is easier to compare different queueing disciplines with a common factor of service time.

First Come First Served

All results are from Jorma Virtamo's lectures on the matter [4].

1. The first is expected waiting time.

   $$E[W] = \frac{(1+C^2)}{2}\frac{\rho}{(1-\rho)} * E[S]$$

   What does this say? When given as $\rho \to 1$ or the mean job arrival rate equals the mean job processing rate, then the wait times get long. Also, as the variance of the job increases, the wait times go up.

2. Next is the expected response time

   $$E[R] = E[N] * E[S] = \lambda * E[W] * E[S]$$

   The response time is simple to calculate, it is the expected number of people ahead of the process in the queue times the expected time to service each of those processes. From Little's Law above, we can substitute that for this. Since we already know the value of the waiting time, we can reason about the response time as well.

3. A discussion of the results is shows something cool discovered by Conway and Al [1]. Any scheduling discipline that isn't preemptive and doesn't take into account the run time of the process or a priority will have the same wait, response, and turnaround time. We will often use this as a baseline.

Round Robin or Processor Sharing

It is hard to analyze Round Robin from a probabilistic sense because it is so state based. The next job that the scheduler schedules requires it to remember the previous jobs. Queueing theory developers have made an assumption that the time quanta is roughly zero – ignoring context switching and the like. This leads way into processor sharing. Many different tasks can get worked on at the same time but experience a slowdown. All of these proofs will be adapted from Harchol-Balter's book [2]. We highly recommend checking out the books if you are interested. The proofs are intuitive for people who don't have a background in queueing theory.

1. Before we jump to the answer let's reason about this. With our new-found abstraction, we essentially have an FCFS queue where we are going to be working on each job a little slower than before. Since we are always working on a job

   $$E[W] = 0$$

   Under a non-strict analysis of processor sharing though, the number of time that the scheduler waits is best approximated by the number of times the scheduler need to wait. You'll need $\frac{E[S]}{Q}$ service periods where Q is the quanta, and you'll need about $E[N] * Q$ time in between those periods. Leading to an average time of

   $$E[W] = E[S] * E[N]$$

   The reason this proof is non-rigorous is that we can't assume that there will always be $E[N] * Q$ time on average in between cycles because it depends on the state of the system. This means we need to factor in various variations in processing delay. We also can't use Little's Law in this case because there is no real steady state of the system. Otherwise, we'd be able to prove some weird things.

   Interestingly, we don't have to worry about the convoy effect or any new processes coming in. The total wait time remains bounded by the number of people in the queue. For those of you familiar with tail inequalities since processes arrive according to a Poisson distribution, the probability that we'll get many processes drops off exponentially due to Chernoff bounds (all arrivals are independent of other arrivals). Meaning roughly we can assume low variance on the number of processes. As long as the service time is reasonable on average, the wait time will be too.

2. The expected response time is

   $$E[R] = 0$$

   Under strict processor sharing, it is 0 because all jobs are worked on. In practice, the response time is.

   $$E[R] = E[N] * Q$$

   Where Q is the quanta. Using Little's Law again, we can find out that

   $$E[R] = \lambda E[W] * Q$$

3. A different variable is the amount of service time let the service time for processor sharing be defined as S_{PS}. The slowdown is $E[S_{PS}] = \frac{E[S]}{1-\rho}$ Which means as the mean arrival rate equals the mean processing time, then the jobs will take asymptotically as long to finish. In the non-strict analysis of processor sharing, we assume that

   $$E[S_{RR}] = E[S] + Q * \epsilon, \epsilon > 0$$

   ϵ is the amount of time a context switch takes.

4. That naturally leads to the comparison, what is better? The response time is roughly the same comparing the non-strict versions, the wait time is roughly the same, but notice that nothing about the variation of the jobs is put in. That's because RR doesn't have to deal with the convoy effect and any variances associated, otherwise FCFS is faster in a strict sense. It also takes more time for the jobs to finish, but the overall turnaround time is lower under high variance loads.

Non Preemptive Priority

We will introduce the notation that there are k different priorities and $\rho_i > 0$ is the average load contribution for priority i We are constrained by $\sum_{i=0}^{k} \rho_i = \rho$. We will also denote $\rho(x) = \sum_{i=0}^{x} \rho_i$ which is the load contribution for all higher and similar priority processes to x. The last bit of notation is that we will assume that the probability of getting a process of priority i is p_i and naturally $\sum_{j=0}^{k} p_j = 1$

1. If $E[W_i]$ is the wait time for priority i,

$$E[W_x] = \frac{(1+C)}{2} \frac{\rho}{(1-\rho(x)) * (1-\rho(x-1))} * E[S_i]$$

The full derivation is as always in the book. A more useful inequality is that.

$$E[W_x] \leq \frac{1+C}{2} * \frac{\rho}{(1-\rho(x))^2} * E[S_i]$$

because the addition of ρ_x can only increase the sum, decrease the denominator or increase the overall function. This means that if one is priority 0, then a process only need to wait for the other P0 processes which there should be $\rho C/(1-\rho_0)$ P0 processes arrived before to process in FCFS order. Then the next priority has to wait for all the others and so on and so forth.

The expected overall wait time is now

$$E[W] = \sum_{i=0}^{k} E[W_i] * p_i$$

Now that we have notational soup, let's factor out the important terms.

$$\sum_{i=0}^{k} \frac{p_i}{(1-\rho(i))^2}$$

Which we compare with FCFS' model of

$$\frac{1}{1-\rho}$$

In words – you can work this out with experimenting distributions – if the system has a lot of low priority processes who don't contribute a lot to the average load, your average wait time becomes much lower.

2. The average per process response time is

$$E[R_i] = \sum_{j=0}^{i} E[N_j] * E[S_j]$$

Which says that the scheduler needs to wait for all jobs with a higher priority and the same to go before a process can go. Imagine a series of FCFS queues that a process needs to wait your turn. Using Little's Law for different colored jobs and the formula above we can simplify this

$$E[R_i] = \sum_{j=0}^{i} \lambda_j E[W_j] * E[S_j]$$

And we can find the average response time by looking at the distribution of jobs

$$E[R] = \sum_{i=0}^{k} p_i [\sum_{j=0}^{k} \lambda_j E[W_j] * E[S_j]]$$

Meaning that we are tied to wait times and service times of all other processes. If we break down this equation, we see again if we have a lot of high priority jobs that don't contribute a lot to the load then our entire sum goes down. We won't make too many assumptions about the service time for a job because that would interfere with our analysis from FCFS where we left it as an expression.

3. As for a comparison with FCFS in the average case, it usually does better assuming that we have a smooth probability distribution – i.e. the probability of getting any particular priority is zero. In all of our formulas, we still have some probability mass to put on lower priority processes, bringing the expectation down. This statement doesn't hold for all smooth distributions but for most real-world smoothed distributions (which tend to be smooth) they do.

4. This isn't even to mention the idea of utility. Utility means that if we gain an amount of happiness by having certain jobs finish, priority and preemptive priority maximize that while balancing out other measures of efficiency.

Shortest Job First

This is a wonderful reduction to priority. Instead of having discrete priorities, we'll introduce a process that takes S_t time to get serviced. T is the maximum amount of time a process can run for, our processes cannot run infinitely long. That means the following definitions hold, overriding the previous definitions in priority

1. Let

   $$\rho(x) = \int_0^x \rho_u du$$

   Be the average load contribution up to this point.

2. $$\int_0^k p_u du = 1$$

   Probability constraint.

3. Etc, replace all the summations above with integrals

4. The only notational difference is we don't have to make any assumptions about the service times of the jobs because they are denoted by service times subscript, all other analyses are the same.

5. This means if you want low wait times on average compared to FCFS, your distribution needs to be right-skewed.

Preemptive Priority

We will describe priority and SJF's preemptive version in the same section because it is essentially the same as we've shown above. We'll use the same notation as before. We will also introduce an additional term C_i which denotes the variation among a particular class

$$C_i = \frac{var(S_i)}{E[S_i]}$$

1. Response Time. Just a head's up, this isn't going to be pretty.

$$E[R_i] = \frac{\sum_{j=0}^{i} \frac{(1+C_j)}{2}}{(1-\rho(x)) * (1-\rho(x-1))} * E[S_i]$$

   If this looks familiar it should. This is the average wait time in the nonpreemptive case with a small change. Instead of using the variance of the entire distribution, we are looking at the variance of each job coming in. The whole response times are

$$E[R] = \sum_{i=0}^{k} p_i * E[R_i]$$

   If lower priorities jobs come in at a higher service time variance, that means our average response times could go down, unless they make up most of the jobs that come in. Think of the extreme cases. If 99% of the jobs are high priority and the rest make up the other percent, then the other jobs will get frequently interrupted, but high priority jobs will make up most of the jobs, so the expectation is still low. The other extreme is if one percent of jobs are high priority and they come in a low variance. That means the chances the system getting a high priority jobs that will take a long time is low, thus making our response times lower on average. We only run into trouble if high priority jobs make up a non-negligible amount, and they have a high variance in service times. This brings down response times as well as wait times.

2. Waiting Time

$$E[W_i] = E[R_i] + \frac{E[S_i]}{1-\rho(i)}$$

   Taking the expectation among all processes we get

$$E[W] = \sum_{i=0}^{k} p_i(E[R_i] + \frac{E[S_i]}{1-\rho(i)})$$

   We can simplify to

$$E[W] = E[R] + \sum_{i=0}^{k} \frac{E[S_i]p_i}{(1-\rho(i))}$$

   We incur the same cost on response time and then we have to suffer an additional cost based on what the probabilities are of lower priority jobs coming in and taking this job out. That is what we call the average interruption time. This follows the same laws as before. Since we have a variadic, pyramid summation if we have a lot of jobs with small service times then the wait time goes down for both additive pieces. It can be analytically shown that this is better given certain probability distributions. For example, try with the uniform versus FCFS or the non preemptive version. What happens? As always the proof is left to the reader.

3. Turnaround Time is the same formula $E[T] = E[S] + E[W]$. This means that given a distribution of jobs that has either low waiting time as described above, we will get low turnaround time – we can't control the distribution of service times.

Preemptive Shortest Job First

Unfortunately, we can't use the same trick as before because an infinitesimal point doesn't have a controlled variance. Imagine the comparisons though as the same as the previous section.

Topics

- Scheduling Algorithms
- Measures of Efficiency

Questions

- What is scheduling?
- What is queueing? What are some different queueing methods?
- What is Turnaround Time? Response Time? Wait Time?
- What is the convoy effect?
- Which algorithms have the best turnaround/response/wait time on average?
- Do preemptive algorithms do better on average response time compared to non preemptive? How about turnaround/wait time?

Bibliography

[1] R.W. Conway, W.L. Maxwell, and L.W. Miller. *Theory of scheduling*. Addison-Wesley Pub. Co., 1967. URL https://books.google.com/books?id=CSozAAAAMAAJ.

[2] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge University Press, 2013. ISBN 9781107027503. URL https://books.google.com/books?id=75SbigDGK0kC.

[3] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 2005. ISBN 9780471694663. URL https://books.google.com/books?id=FH8fAQAAIAAJ.

[4] Jorma Virtamo. 38.3143 queueing theory / them/g/1/queue. URL https://www.netlab.tkk.fi/opetus/s383143/kalvot/E_mg1jono.pdf.

11
Networking

> The Web as I envisaged it, we have not seen it yet. The future is still so much bigger than the past
>
> Tim Berners-Lee

Networking has become arguably the most important use of computers in the past 10-20 years. Most of us nowadays can't stand a place without WiFi or any connectivity, so it is crucial as programmers that you have an understanding of networking and how to program to communicate across networks. Although it may sound complicated, POSIX has defined nice standards that make connecting to the outside world easy. POSIX also lets you peer underneath the hood and optimize all the little parts of each connection to write highly performant programs.

As an addendum that you'll read more about in the next chapter, we will be strict in our notation for sizes. That means that when we refer to the SI prefixes of Kilo-, Mega-, etc, then we are always referring to a power of 10. A kilobyte is one thousand bytes, a megabyte is a thousand kilobytes and so on. If we need to refer to 1024 bytes, we will use the more accurate term Kibibyte. Mibibyte and Gibibyte are the analogs of Megabyte and Gigabyte respectively. We make this distinction to make sure that we aren't off by 24. The reasons for this misnomer will be explained in the filesystems chapter.

The OSI Model

The Open Source Interconnection 7 layer model (OSI Model) is a sequence of segments that define standards for both infrastructure and protocols for forms of radio communication, in our case the Internet. The 7 layer model is as follows

1. Layer 1: The Physical Layer. These are the actual waves that carry the bauds across the wire. As an aside, bits don't cross the wire because in most mediums you can alter two characteristics of a wave – the amplitude and the frequency – and get more bits per clock cycle.

2. Layer 2: The Link Layer. This is how each of the agents reacts to certain events (error detection, noisy channels, etc). This is where Ethernet and WiFi live.

3. Layer 3: The Network Layer. This is the heart of the Internet. The bottom two protocols deal with communication between two different computers that are directly connected. This layer deals with routing packets from one endpoint to another.

4. Layer 4: The Transport Layer. This layer specifies how the slices of data are received. The bottom three layers make no guarantee about the order that packets are received and what happens when a packet is dropped. Using different protocols, this layer can.

5. Layer 5: The Session Layer. This layer makes sure that if a connection in the previous layers is dropped, a new connection in the lower layers can be established, and it looks like nothing happened to the end-user.

6. Layer 6: The Presentation Layer. This layer deals with encryption, compression, and data translation. For example, portability between different operating systems like translating newlines to windows newlines.

7. Layer 7: The Application Layer. Hyper Text Transfer Protocol and File Transfer Protocol are both defined at this level. This is typically where we define protocols across the Internet. As programmers, we only go lower when we think we can create algorithms that are more suited to our needs than all of the below.

This book won't cover networking in depth. We will focus on some aspects of layers 3, 4, and 7 because they are essential to know if you are going to be doing something with the Internet, which at some point in your career you will be. As for another definition, a protocol is a set of specifications put forward by the Internet Engineering Task Force that govern how implementers of a protocol have their program or circuit behave under specific circumstances.

Layer 3: The Internet Protocol

The following is a short introduction to internet protocol (IP), the primary way to send datagrams of information from one machine to another. "IP4", or more precisely, Internet Protocol Version 4 is version 4 of the Internet Protocol that describes how to send Packet of information across a network from one machine to another. Even as of 2018, IPv4 still dominates Internet traffic, but google reports that 24 countries now supply 15% of their traffic through IPv6 [2]. A significant limitation of IPv4 is that source and destination addresses are limited to 32 bits. IPv4 was designed at a time when the idea of 4 billion devices connected to the same network was unthinkable or at least not worth making the packet size larger. are written typically in a sequence of four octets delimited by periods "255.255.255.0" for example.

Each IPv4 Datagram includes a small header - typically 20 , that includes a source and destination address. Conceptually the source and destination addresses can be split into two: a network number the upper bits and lower bits represent a particular host number on that network.

A newer packet protocol Internet Protocol Version 6 solves many of the limitations of IPv4 like making routing tables simpler and 128-bit addresses. However, little web traffic is IPv6 based on comparison as of 2018 [2] We write IPv6 addresses in a sequence of eight, four hexadecimal delimiters like "1F45:0000:0000:0000:0000:0000:0000:0000". Since that can get unruly, we can omit the zeros "1F45::". A machine can have an IPv6 address and an IPv4 address.

There are special IP Addresses. One such in IPv4 is `127.0.0.1`, IPv6 as `0:0:0:0:0:0:0:1` or `::1` also known as localhost. Packets sent to 127.0.0.1 will never leave the machine; the address is specified to be the same machine. There are a lot of others that are denoted by certain octets being zeros or 255, the maximum value. You won't need to know all the terminology, keep in mind that the actual number of IP addresses that a machine can have globally over the Internet is smaller than the number of "raw" addresses. This book covers how IP deals with routing, fragmenting, and reassembling upper-level protocols. A more in-depth aside follows.

Extra: In-depth IPv4 Specification

The Internet Protocol deals with routing, fragmentation, and reassembly of fragments. Datagrams are formatted as such

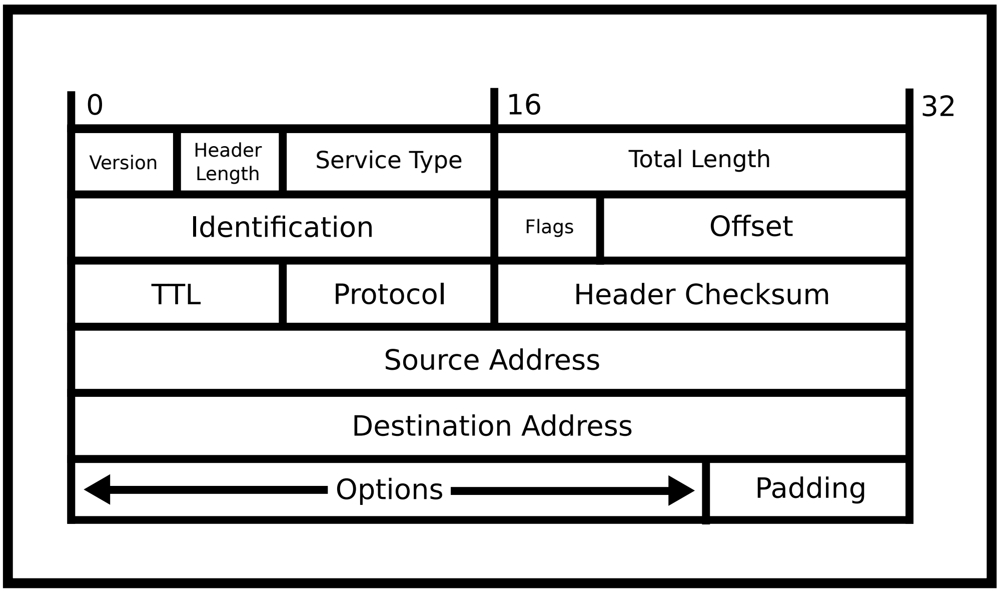


Figure 11.1: IP Datagram divisibility

1. The first octet is the version number, either 4 or 6

2. The next octet is how long the header is. Although it may seem that the header is a constant size, you can include optional parameters to augment the path that is taken or other instructions.

3. The next two octets specify the total length of the datagram. This means this is the header, the data, the footer, and the padding. This is given in multiple of octets, meaning that a value of 20 means 20 octets.

4. The next two are Identification number. IP handles taking packets that are too big to be sent over the physical wire and chunks them up. As such, this number identifies what datagram this originally belonged to.

5. The next octet is various bit flags that can be set.

6. The next octet and half is fragment number. If this packet was fragmented, this is the number this fragment represents

7. The next octet is time to live. So this is the number of "hops" (travels over a wire) a packet is allowed to go. This is set because different routing protocols could cause packets to go in circles, the packets must be dropped at some point.

8. The next octet is the protocol number. Although protocols between different layers of the OCI model are supposed to be black boxes, this is included, so that hardware can peer into the underlying protocol efficiently. Take for example IP over IP (yes you can do that!). Your ISP wraps IPv4 packets sent from your computer to the ISP in another IP layer and sends the packet off to be delivered to the website. On the reverse trip, the packet is "unwrapped" and the original IP datagram is sent to your computer. This was done because we ran out of IP addresses, and this adds additional overhead but it is a necessary fix. Other common protocols are TCP, UDP, etc.

9. The next two octets is an internet checksum. This is a CRC that is calculated to make sure that a wide variety of bit errors are detected.

10. The source address is what people generally refer to as the IP address. There is no verification of this, so one host can pretend to be any IP address possible

11. The destination address is where you want the packet to be sent to. Destinations are crucial to the routing process.

12. Additional options: Hosts of additional options, this is variadic in size.

13. Footer: A bit of padding to make sure your data is a multiple of 4 octets.

14. After: Your data! All data of higher-order protocols are put following the header.

Extra: Routing

The Internet Protocol routing is an amazing intersection of theory and application. We can imagine the entire Internet as a set of graphs. Most peers are connected to what we call "peering points" – these are the WiFi routers and Ethernet ports that one finds at home, at work, and in public. These peering points are then connected to a wired network of routers, switches, and servers that all route themselves. At a high level there are two types of routing

1. Internal Routing Protocols. Internal protocols are routing designed for within an ISP's network. These protocols are meant to be fast and more trusting because all computers, switches, and routers are part of an ISP. communication between two routers.

2. External Routing Protocols. These typically happen to be ISP to ISP protocol. Certain routers are designated as border routers. These routers talk to routers from ISPs who have different policies from accepting or receiving packets. If an evil ISP is trying to dump all network traffic onto your ISP, these routers would deal with that. These protocols also deal with gathering information about the outside world to each router. In most routing protocols using link state or OSPF, a router must necessarily calculate the shortest path to the destination. This means it needs information about the "foreign" routers which is disseminated according to these protocols.

These two protocols have to interplay with each other nicely to make sure that packets are mostly delivered. Also, ISPs need to be nice to each other. Theoretically, an ISP can handle a smaller load by forwarding all packets to another ISP. If everyone does that then, no packets get delivered at all which won't make customers happy at all. These two protocols need to be fair so the result works

If you want to read more about this, look at the Wikipedia page for routing here Routing.

Extra: Fragmentation/Reassembly

Lower layers like WiFi and Ethernet have maximum transmission sizes. The reason being is

1. One host shouldn't crowd the medium for too long

2. If an error occurs, we want some sort of "progress bar" on how far the communication has gone instead of retransmitting the entire stream.

3. There are physical limitations, keeping a laser beam in optics working continuously may cause bit errors.

If the Internet Protocol receives a packet that is too big for the maximum size, it must chunk it up. TCP calculates how many datagrams that it needs to construct a packet and ensures that they are all transmitted and reconstructed at the end receiver. The reason that we barely use this feature is that if any fragment is lost, the entire packet is lost. Meaning that, assuming the probability of receiving a packet assuming each fragment is lost with an independent percentage, the probability of successfully sending a packet drops off exponentially as packet size increases.

As such, TCP slices its packets so that it fits inside on IP datagram. The only time that this applies is when sending UDP packets that are too big, but most people who are using UDP optimize and set the same packet size as well.

Extra: IP Multicast

A little known feature is that using the IP protocol one can send a datagram to all devices connected to a router in what is called a multicast. Multicasts can also be configured with groups, so one can efficiently slice up all connected routers and send a piece of information to all of them efficiently. To access this in a higher protocol, you need to use UDP and specify a few more options. Note that this will cause undue stress on the network, so a series of multicasts could flood the network fast.

What's the deal with IPv6?

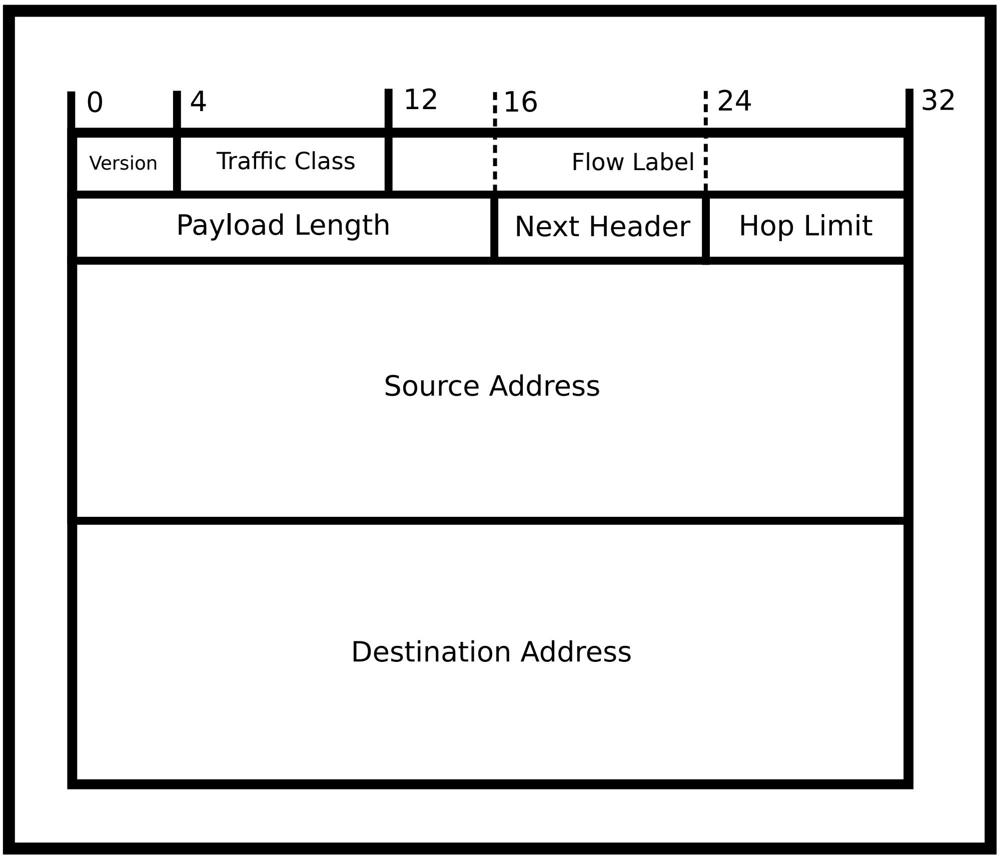


Figure 11.2: IPv6 Datagram divisibility

One of the big features of IPv6 is the address space. The world ran out of IP addresses a while ago and has been using hacks to get around that. With IPv6 there are enough internal and external addresses so even if we discover alien civilizations, we probably won't run out. The other benefit is that these addresses are leased not bought, meaning that if something drastic happens in let's say the Internet of things and there needs to be a change in the block addressing scheme, it can be done.

Another big feature is security through IPsec. IPv4 was designed with little to no security in mind. As such, now there is a key exchange similar to TLS in higher layers that allows you to encrypt communication.

Another feature is simplified processing. To make the Internet fast, IPv4 and IPv6 headers are verified in hardware. That means that all header options are processed in circuits as they come in. The problem is that as the IPv4 spec grew to include a copious amount of headers, the hardware had to become more and more advanced to support those headers. IPv6 reorders the headers so that packets can be dropped and routed with fewer hardware cycles. In the case of the Internet, every cycle matters when trying to route the world's traffic.

What's My Address?

To obtain a linked list of IP addresses of the current machine use getifaddrs which will return a linked list of IPv4 and IPv6 IP addresses among other interfaces as well. We can examine each entry and use getnameinfo to print the host's IP address. The ifaddrs struct includes the family but does not include the sizeof the struct. Therefore we need to manually determine the struct sized based on the family.

```
(family == AF_INET) ? sizeof(struct sockaddr_in) : sizeof(struct
    sockaddr_in6)
```

The complete code is shown below.

```
int required_family = AF_INET; // Change to AF_INET6 for IPv6
struct ifaddrs *myaddrs, *ifa;
getifaddrs(&myaddrs);
char host[256], port[256];

for (ifa = myaddrs; ifa != NULL; ifa = ifa->ifa_next) {
  int family = ifa->ifa_addr->sa_family;
  if (family == required_family && ifa->ifa_addr) {
    int ret = getnameinfo(ifa->ifa_addr,
    (family == AF_INET) ? sizeof(struct sockaddr_in) :
    sizeof(struct sockaddr_in6),
    host, sizeof(host), port, sizeof(port)
    , NI_NUMERICHOST | NI_NUMERICSERV)
    if (0 == ret) {
      puts(host);
    }
  }
}
```

To get your IP Address from the command line use ifconfig or Windows' ipconfig.
However, this command generates a lot of output for each interface, so we can filter the output using grep.

```
ifconfig | grep inet

Example output:
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    inet 127.0.0.1 netmask 0xff000000
    inet6 ::1 prefixlen 128
    inet6 fe80::7256:81ff:fe9a:9141%en1 prefixlen 64 scopeid 0x5
    inet 192.168.1.100 netmask 0xffffff00 broadcast 192.168.1.255
```

To grab the IP Address of a remote website, The function getaddrinfo can convert a human-readable domain name (e.g. www.illinois.edu) into an IPv4 and IPv6 address. It will return a linked-list of addrinfo structs:

```
struct addrinfo {
  int              ai_flags;
  int              ai_family;
  int              ai_socktype;
  int              ai_protocol;
  socklen_t        ai_addrlen;
  struct sockaddr *ai_addr;
  char            *ai_canonname;
  struct addrinfo *ai_next;
};
```

For example, suppose you wanted to find out the numeric IPv4 address of a web server at www.bbc.com. We do this in two stages. First, use getaddrinfo to build a linked-list of possible connections. Secondly, use getnameinfo to convert the binary address of one of those into a readable form.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

struct addrinfo hints, *infoptr; // So no need to use memset
    global variables

int main() {
  hints.ai_family = AF_INET; // AF_INET means IPv4 only addresses

  // Get the machine addresses
  int result = getaddrinfo("www.bbc.com", NULL, &hints, &infoptr);
  if (result) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
    exit(1);
  }

  struct addrinfo *p;
  char host[256];

  for(p = infoptr; p != NULL; p = p->ai_next) {
    // Get the name for all returned addresses
    getnameinfo(p->ai_addr, p->ai_addrlen, host, sizeof(host),
        NULL, 0, NI_NUMERICHOST);
    puts(host);
  }

  freeaddrinfo(infoptr);
  return 0;
}
```

Possible output.

```
212.58.244.70
212.58.244.71
```

One can specify IPv4 or IPv6 with AF_UNSPEC. Just replace the ai_family attribute in the above code with the following.

```
hints.ai_family = AF_UNSPEC
```

If you are wondering how the computer maps Hostname to addresses, we will talk about that in Layer 7. Spoiler: It is a service called Domain Name Service. Before we move onto the next section, it is important to note that a single website can have multiple IP addresses. This may be to be efficient with machines. If Google or Facebook has a single server routing *all* of their incoming requests to other computers, they'd have to spend massive amounts of money on that computer or data center. Instead, they can give different regions different IP addresses and have a computer pick. It isn't bad to access a website through the non-preferred IP address. The page may load slower.

Layer 4: TCP and Client

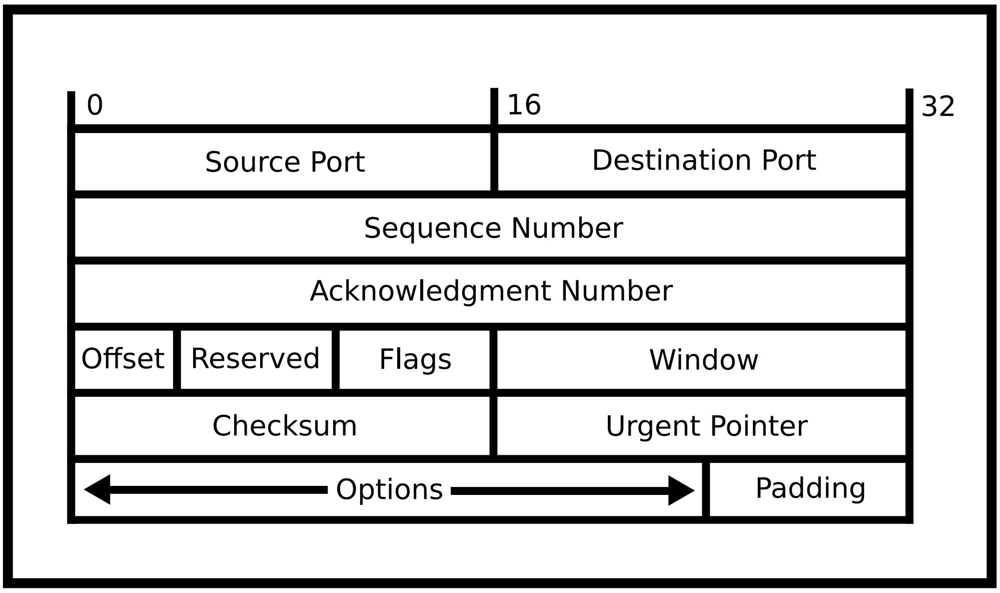


Figure 11.3: Extra: TCP Header Specification

Most services on the Internet today use Transport Control Protocol because it efficiently hides the complexity of the lower, packet-level nature of the Internet. TCP or Transport Control Protocol is a connection-based protocol that is built on top of IPv4 and IPv6 and therefore can be described as "TCP/IP" or "TCP over IP". TCP creates a *pipe* between two machines and abstracts away the low-level packet-nature of the Internet. Thus, under most conditions, bytes sent over a TCP connection delivered and uncorrupted. High performance and error-prone code won't even assume that!

TCP has many features that set it apart from the other transport protocol UDP.

1. Port With IP, you are only allowed to send packets to a machine. If you want one machine to handle multiple flows of data, you have to do it manually with IP. TCP gives the programmer a set of virtual sockets. Clients specify the socket that you want the packet sent to and the TCP protocol makes sure that applications that are waiting for packets on that port receive that. A process can listen for incoming packets on a particular port. However, only processes with (root) access can listen on ports less than 1024. Any process can listen on ports 1024 or higher. A frequently used port is number 80. It is used for unencrypted HTTP requests or web pages. For example, if a web browser connects to `http://www.bbc.com/` then it will be connecting to port 80.

2. Packets can get dropped due to network errors or congestion. As such, they need to be retransmitted. At the same time, the retransmission shouldn't cause packets more packets to be dropped. This needs to balance the tradeoff between flooding the network and speed.

3. Out of order packets. Packets may get routed more favorably due to various reasons in IP. If a later packet arrives before another packet, the protocol should detect and reorder them.

4. Duplicate packets. Packets can arrive twice. Packets can arrive twice. As such, a protocol needs to be able to differentiate between two packets given a sequence number subject to overflow.

5. Error correction. There is a TCP checksum that handles bit errors. This is rarely used though.
6. Flow Control. Flow control is performed on the receiver side. This may be done so that a slow receiver doesn't get overwhelmed with packets. Servers that handle 10000 or 10 million concurrent connections may need to tell receivers to slow down but remain connected due to load. There is also the problem of making sure the local network's traffic is stable.
7. Congestion control. Congestion control is performed on the sender's side. Congestion control is to avoid a sender from flooding the network with too many packets. This is important to make sure that each TCP connection is treated fairly. Meaning that two connections leaving a computer to google and youtube receive the same bandwidth and ping as each other. One can easily define a protocol that takes all the bandwidth and leaves other protocols in the dust, but this tends to be malicious because many times limiting a computer to a single TCP connection will yield the same result.
8. Connection-Oriented/life cycle oriented. You can imagine a TCP connection as a series of bytes sent through a pipe. There is a "lifecycle" to a TCP connection though. TCP handles setting up the connection through SYN SYN-ACK ACK. This means the client will send a SYNchronization packet that tells TCP what starting sequence to start on. Then the receiver will send a SYN-ACK message acknowledging the synchronization number. Then the client will ACKnowledge that with one last packet. The connection is now open for both reading and writing on both ends TCP will send data and the receiver of the data will acknowledge that it received a packet. Then every so often if a packet is not sent, TCP will trade zero-length packets to make sure the connection is still alive. At any point, the client and server can send a FIN packet meaning that the server will not transmit. This packet can be altered with bits that only close the read or write end of a particular connection. When all ends are closed then the connection is over.

TCP doesn't provide many things, though.

1. Security. Connecting to an IP address claiming to be a certain website does not verify the claim (like in TLS). You could be sending packets to a malicious computer.
2. Encryption. Anybody can listen in on plain TCP. The packets in transport are in plain text. Important things like your passwords could easily be skimmed by onlookers.
3. Session Reconnection. If a TCP connection dies then a whole new one must be created, and the transmission has to be started over again. This is handled by a higher protocol.
4. Delimiting Requests. TCP is naturally connection-oriented. Applications that are communicating over TCP need to find a unique way of telling each other that this request or response is over. HTTP delimits the header through two carriage returns and uses either a length field or one keeps listening until the connection closes

Note on network orders

Integers can be represented in the least significant byte first or most significant byte first. Either approach is reasonable as long as the machine itself is internally consistent. For network communications, we need to standardize on the agreed format.

`htons(xyz)` returns the 16-bit unsigned integer 'short' value xyz in network byte order. `htonl(xyz)` returns the 32-bit unsigned integer 'long' value xyz in network byte order. Any longer integers need to have the computers specify the order.

These functions are read as 'host to network'. The inverse functions (`ntohs`, `ntohl`) convert network ordered byte values to host-ordered ordering. So, is host-ordering little-endian or big-endian? The answer is - it depends on your machine! It depends on the actual architecture of the host running the code. If the architecture happens to be the same as network ordering then the functions return identical integers. For x86 machines, the host and network order are different.

Unless agreed otherwise, whenever you read or write the low-level C network structures, i.e. port and address information, remember to use the above functions to ensure correct conversion to/from a machine format. Otherwise, the displayed or specified value may be incorrect.

This doesn't apply to protocols that negotiate the endianness before-hand. If two computers are CPU bound by converting the messages between network orders – this happens with RPCs in high-performance systems – it may be worth it to negotiate if they are on similar endianness to send in little-endian order.

Why is network order defined to be big-endian? The simple answer is that RFC1700 says so [5]. If you want more information, we'll cite the famous article located that argued for a particular version [3]. The most important part is that it is standard. What happens when we don't have one standard? We have 4 different USB plug types (Regular, Micro, Mini, and USB-C) that don't interact well with each other. Include relevant XKCD here Standards.

TCP Client

There are three basic system calls to connect to a remote machine.

1. `int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct`

   The `getaddrinfo` call if successful, creates a linked-list of `addrinfo` structs and sets the given pointer to point to the first one.

   Also, you can use the hints struct to only grab certain entries like certain IP protocols, etc. The addrinfo structure that is passed into `getaddrinfo` to define the kind of connection you'd like. For example, to specify stream-based protocols over IPv6, you can use the following snippet.

```
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));

hints.ai_family = AF_INET6; // Only want IPv6 (use AF_INET for
    IPv4)
hints.ai_socktype = SOCK_STREAM; // Only want stream-based
    connection
```

   The other modes for 'family' are `AF_INET4` and `AF_UNSPEC` which mean IPv4 and unspecified respectively. This could be useful if you are searching for a service that you aren't entirely sure which IP version. Naturally, you get the version in the field back if you specified UNSPEC.

   Error handling with `getaddrinfo` is a little different. The return value *is* the error code. To convert to a human-readable error use `gai_strerror` to get the equivalent short English error text.

```
int result = getaddrinfo(...);
if(result) {
  const char *mesg = gai_strerror(result);
  ...
}
```

2. int socket(int domain, int socket_type, int protocol);

   The socket call creates a network socket and returns a descriptor that can be used with read and write. In this sense, it is the network analog of open that opens a file stream – except that we haven't connected the socket to anything yet!

   Sockets are created with a domain AF_INET for IPv4 or AF_INET6 for IPv6, socket_type is whether to use UDP, TCP, or other some other socket type, the protocol is an optional choice of protocol configuration for our examples this we can leave this as 0 for default. This call creates a socket object in the kernel with which one can communicate with the outside world/network. You can use the result of getaddressinfo to fill in the socket parameters, or provide them manually.

   The socket call returns an integer - a file descriptor - and, for TCP clients, you can use it as a regular file descriptor. You can use read and write to receive or send packets.

   TCP sockets are similar to pipes and are often used in situations that require IPC. We don't mention it in the previous chapters because it is overkill using a device suited for networks to simply communicate between processes on a single thread.

3. connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

   Finally, the connect call attempts the connection to the remote machine. We pass the original socket descriptor and also the socket address information which is stored inside the addrinfo structure. There are different kinds of socket address structures that can require more memory. So in addition to passing the pointer, the size of the structure is also passed. To help identify errors and mistakes it is good practice to check the return value of all networking calls, including connect

```
// Pull out the socket address info from the addrinfo struct:
connect(sockfd, p->ai_addr, p->ai_addrlen)
```

4. (Optional) To clean up code call freeaddrinfo(struct addrinfo *ai) on the first level addrinfo struct.

There is an old function gethostbyname is deprecated. It's the old way convert a hostname into an IP address. The port address still needs to be manually set using htons function. It's much easier to write code to support IPv4 AND IPv6 using the newer getaddrinfo

This is all that is needed to create a *simple* TCP client. However, network communications offer many different levels of abstraction and several attributes and options that can be set at each level. For example, we haven't talked about setsockopt which can manipulate options for the socket. You can also mess around with lower protocols as the kernel provides primitives that contribute to this. Note that you need to be root to create a raw socket. Also, you need to have a lot of "set up" or starter code, be prepared to have your datagrams be dropped due to bad form as well. For more information see this guide.

Sending some data

Once we have a successful connection we can read or write like any old file descriptor. Keep in mind if you are connected to a website, you want to conform to the HTTP protocol specification to get any sort of meaningful results back. There are libraries to do this. Usually, you don't connect at the socket level. The number of bytes read or written may be smaller than expected. Thus, it is important to check the return value of read and write. A simple HTTP client that sends a request to a compliant URL is below. First, we'll start with the boring stuff and the parsing code.

```c
typedef struct _host_info {
  char *hostname;
  char *port;
  char *resource;
} host_info;

host_info *get_info(char *uri) {
  // ... Parses the URI/URL
}

void free_info(host_info *info) {
  // ... Frees any info
}

int main(int argc, char *argv[]) {
  if(argc != 2) {
    fprintf(stderr, "Usage: %s http://hostname[:port]/path\n",
        *argv);
    return 1;
  }
  char *uri = argv[1];
  host_info *info = get_info(uri);
  host_info *temp = send_request(info);

  return 0;
}
```

The code that sends the request is below. The first thing that we have to do is connect to an address.

```c
struct addrinfo current, *result;
memset(&current, 0, sizeof(struct addrinfo));
current.ai_family = AF_INET;
current.ai_socktype = SOCK_STREAM;

getaddrinfo(info->hostname, info->port, &current, &result);
```

```
connect(sock_fd, result->ai_addr, result->ai_addrlen)

freeaddrinfo(result);
```

The next piece of code sends the request. Here is what each header means.

1. "GET %s HTTP/1.0" This is the request verb interpolated with the path. This means to perform the GET verb on the path using the HTTP/1.0 method.
2. "Connection: close" Means that as soon as the request is over, please close the connection. This line won't be used for any other connections. This is a little redundant given that HTTP 1.0 doesn't allow you to send multiple requests, but it is better to be explicit given there are non-conformant technologies.
3. "Accept: */*" This means that the client is willing to accept anything.

A more robust piece of code would also check if the write fails or if the call was interrupted.

```
char *buffer;
asprintf(&buffer,
  "GET %s HTTP/1.0\r\n"
  "Connection: close\r\n"
  "Accept: */*\r\n\r\n",
  info->resource);

write(sock_fd, buffer, strlen(buffer));
free(buffer);
```

The last piece of code is the driver code that sends the request. Feel free to use the following code if you want to open the file descriptor as a FILE object for convenience functions. Just be careful not to forget to set the buffering to zero otherwise you may double buffer the input, which would lead to performance problems.

```
void send_request(host_info *info) {
  int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
  // Re-use address is a little overkill here because we are making
      a
  // Listen only server and we don't expect spoofed requests.
  int optval = 1;
  int retval = setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR,
      &optval,
  sizeof(optval));
  if(retval == -1) {
    perror("setsockopt");
    exit(1);
  }
  // Connect using code snippet
  // Send the get request

  // Open so you can use getline
  FILE *sock_file = fdopen(sock_fd, "r+");
  setvbuf(sock_file, NULL, _IONBF, 0);

  ret = handle_okay(sock_file);
  fclose(sock_file);
  close(sock_fd);
}
```

The example above demonstrates a request to the server using the HyperText Transfer Protocol. In general, there are six parts

1. The method. GET, POST, etc.
2. The resource. "/" "/index.html" "/image.png"
3. The protocol "HTTP/1.0"
4. A new line (rn). Requests always have a carriage return.
5. Any other knobs or switch parameters
6. The actual body of the request delimited by two new lines. The body of the request is either if the size is specified or until the receiver closes their connection.

The server's first response line describes the HTTP version used and whether the request is successful using a 3 digit response code.

```
HTTP/1.1 200 OK
```

If the client had requested a non-existent path, e.g. `GET /nosuchfile.html HTTP/1.0` Then the first line includes the response code is the well-known 404 response code.

```
HTTP/1.1 404 Not Found
```

For more information, RFC 7231 has the most current specifications on the most common HTTP method today [4].

Layer 4: TCP Server

The four system calls required to create a minimal TCP server are `socket`, `bind`, `listen`, and `accept`. Each has a specific purpose and should be called in roughly the above order

1. `int socket(int domain, int socket_type, int protocol)`

   To create an endpoint for networking communication. A new socket by itself is stores bytes. Though we've specified either a packet or stream-based connections, it is unbound to a particular network interface or port. Instead, socket returns a network descriptor that can be used with later calls to bind, listen and accept.

   As one gotcha, these sockets must be declared passive. Passive server sockets wait for another host to connect. Instead, they wait for incoming connections. Additionally, server sockets remain open when the peer disconnects. Instead, the client communicates with a separate active socket on the server that is specific to that connection.

   Since a TCP connection is defined by the sender address and port along with a receiver address and port, a particular server port there can be one passive server socket but multiple active sockets. One for each currently open connection. The server's operating system maintains a lookup table that associates a unique tuple with active sockets so that incoming packets can be correctly routed to the correct socket.

2. `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`

   The `bind` call associates an abstract socket with an actual network interface and port. It is possible to call bind on a TCP client. The port information used by bind can be set manually (many older IPv4-only C code examples do this), or be created using `getaddrinfo`.

   By default, a port is released after some time when the server socket is closed. Instead, the port enters a "TIMED-WAIT" state. This can lead to significant confusion during development because the timeout can make valid networking code appear to fail.

   To be able to immediately reuse a port, specify `SO_REUSEPORT` before binding to the port.

   ```
   int optval = 1;
   setsockopt(sfd, SOL_SOCKET, SO_REUSEPORT, &optval,
      sizeof(optval));

   bind(...);
   ```

   Here's an extended stackoverflow introductory discussion of `SO_REUSEPORT`.

3. `int listen(int sockfd, int backlog);`

   The `listen` call specifies the queue size for the number of incoming, unhandled connections. There are the connections unassigned to a file descriptor by `accept`. Typical values for a high-performance server are 128 or more.

4. `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`

   Once the server socket has been initialized the server calls `accept` to wait for new connections. Unlike `socket` `bind` and `listen`, this call will block, unless the nonblocking option has been set. If there are no new connections, this call will block and only return when a new client connects. The returned TCP socket is associated with a particular tuple `(client IP, client port, server IP, server port)` and will be used for all future incoming and outgoing TCP packets that match this tuple.

   Note the `accept` call returns a new file descriptor. This file descriptor is specific to a particular client. It is a common programming mistake to use the original server socket descriptor for the server I/O and then wonder why networking code has failed.

The `accept` system call can optionally provide information about the remote client, by passing in a sockaddr struct. Different protocols have different variants of the `struct sockaddr`, which are different sizes. The simplest struct to use is the `sockaddr_storage` which is sufficiently large to represent all possible types of sockaddr. Notice that C does not have any model of inheritance. Therefore we need to explicitly cast our struct to the 'base type' struct sockaddr.

```
struct sockaddr_storage clientaddr;
socklen_t clientaddrsize = sizeof(clientaddr);
int client_id = accept(passive_socket,
  (struct sockaddr *) &clientaddr,
  &clientaddrsize);
```

We've already seen `getaddrinfo` that can build a linked list of addrinfo entries and each one of these can include socket configuration data. What if we wanted to turn socket data into IP and port addresses? Enter `getnameinfo` that can be used to convert local or remote socket information into a domain name or numeric IP. Similarly, the port number can be represented as a service name. For example, port 80 is commonly used as the incoming connection port for incoming HTTP requests. In the example below, we request numeric versions for the client IP address and client port number.

```
socklen_t clientaddrsize = sizeof(clientaddr);
int client_id = accept(sock_id, (struct sockaddr *)
    &clientaddr, &clientaddrsize);
char host[NI_MAXHOST], port[NI_MAXSERV];
getnameinfo((struct sockaddr *) &clientaddr,
 clientaddrsize, host, sizeof(host), port, sizeof(port),
 NI_NUMERICHOST | NI_NUMERICSERV);
```

One can use the macros `NI_MAXHOST` to denote the maximum length of a hostname, and `NI_MAXSERV` to denote the maximum length of a port. `NI_NUMERICHOST` gets the hostname as a numeric IP address and similarly for `NI_NUMERICSERV` although the port is usually numeric, to begin with. The Open BSD man pages have more information

5. `int close(int fd)` and `int shutdown(int fd, int how)`

   Use the `shutdown` call when you no longer need to read any more data from the socket, write more data, or have finished doing both. When you call `shutdown` on socket on the read and/or write ends, that information is also sent to the other end of the connection. If you shut down the socket for further writing at the server end, then a moment later, a blocked `read` call could return 0 to indicate that no more bytes are expected. Similarly, a write to a TCP connection that has been shut down for reading will generate a SIGPIPE

   Use `close` when your process no longer needs the socket file descriptor.

   If you `fork`-ed after creating a socket file descriptor, all processes need to close the socket before the socket resources can be reused. If you shut down a socket for further read, all processes are affected because you've changed the socket, not the file descriptor. Well written code will `shutdown` a socket before calling `close` it.

There are a few gotchas to creating a server.

- Using the socket descriptor of the passive server socket (described above)
- Not specifying SOCK_STREAM requirement for getaddrinfo
- Not being able to reuse an existing port.
- Not initializing the unused struct entries
- The bind call will fail if the port is currently in use. Ports are per machine – not per process or user. In other words, you cannot use port 1234 while another process is using that port. Worse, ports are by default 'tied up' after a process has finished.

Example Server

A working simple server example is shown below. Note: this example is incomplete. For example, the socket file descriptor remains open and memory created by getaddrinfo remains allocated. First, we get the address info for our current machine.

```
struct addrinfo hints, *result;
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

int s = getaddrinfo(NULL, "1234", &hints, &result);
if (s != 0) {
  fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
  exit(1);
}
```

Then we set up the socket, bind it, and listen.

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

// Bind and listen
if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
  perror("bind()");
  exit(1);
}

if (listen(sock_fd, 10) != 0) {
  perror("listen()");
  exit(1);
}
```

We are finally ready to listen for connections, so we'll tell the user and accept our first client.

```
struct sockaddr_in *result_addr = (struct sockaddr_in *)
    result->ai_addr;
printf("Listening on file descriptor %d, port %d\n", sock_fd,
    ntohs(result_addr->sin_port));

// Waiting for connections like a passive socket
printf("Waiting for connection...\n");
int client_fd = accept(sock_fd, NULL, NULL);
printf("Connection made: client_fd=%d\n", client_fd);
```

After that, we can treat the new file descriptor as a stream of bytes much like a pipe.

```
char buffer[1000];
// Could get interrupted
int len = read(client_fd, buffer, sizeof(buffer) - 1);
buffer[len] = '\0';

printf("Read %d chars\n", len);
printf("===\n");
printf("%s\n", buffer);
```

[language=C]

Sorry To Interrupt

One concept that we need to make clear is that you need to handle interrupts in your networking code. That means that the sockets or accepted file descriptors that you read to or write to may have their calls interrupted – most of the time you will get an interrupt or two. In reality, any of your system calls could get interrupted. The reason we bring this up now is that you are usually waiting for the network. Which is an order of magnitude slower than processes. Meaning a higher probability of getting interrupted.

How would you handle interrupts? Let's try a quick example.

```
while bytes_read isn't count {
  bytes_read += read(fd, buf, count);
  if error is EINTR {
    continue;
  } else {
    break;
  }
}
```

We can assure you that the following code *experience errors*. Can you see why? On the surface, it does restart a call after a read or write. But what else happens when the error is EINTR? Are the contents of the buffer correct? What other problems can you spot?

Layer 4: UDP

UDP is a connectionless protocol that is built on top of IPv4 and IPv6. It's simple to use. Decide the destination address and port and send your data packet! However, the network makes no guarantee about whether the packets will arrive. Packets may be dropped if the network is congested. Packets may be duplicated or arrive out of order.

A typical use case for UDP is when receiving up to date data is more important than receiving all of the data. For example, a game may send continuous updates of player positions. A streaming video signal may send picture updates using UDP

UDP Attributes

- Unreliable Datagram Protocol Packets sent through UDP may be dropped on their way to the destination. This can especially be confusing because if you only test on your loop-back device – this is localhost or 127.0.0.1 for most users – then packets will seldom be lost because no network packets are sent.
- Simple The UDP protocol is supposed to have much less fluff than TCP. Meaning that for TCP there are a lot of configurable parameters and a lot of edge cases in the implementation. UDP is fire and forget.
- Stateless/Transaction The UDP protocol is stateless. This makes the protocol more simple and lets the protocol represent simple transactions like requesting or responding to queries. There is also less overhead to sending a UDP message because there is no three-way handshake.
- Manual Flow/Congestion Control You have to manually manage the flow and congestion control which is a double-edged sword. On one hand, you have full control over everything. On the other hand, TCP has *decades* of optimization, meaning your protocol for its use cases needs to be more efficient that to be more beneficial to use it.
- Multicast This is one thing that you can only do with UDP. This means that you can send a message to every peer connected to a particular router that is part of a particular group.

The full gory description is available at the original RFC [1].

While it may seem that you never want to use UDP for situations that you don't want to lose data, a lot of protocols base their communication based on UDP that requires complete data. Take a look at the Trivial File Transfer Protocol that reliably transmits a file over the wire using UDP only. Of course, there is more configuration involved, but choosing between UDP over TCP involves more than the above factors.

UDP Client

UDP Clients are pretty versatile below is a simple client that sends a packet to a server specified through the command line. Note that this client sends a packet and doesn't wait for an acknowledgment. It fires and forgets. The example below also uses gethostbyname because some legacy functionality still works pretty well for setting up a client.

```
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons((uint16_t)port);
struct hostent *serv = gethostbyname(hostname);
```

The previous code grabs an entry hostent that matches by hostname. Even though this isn't portable, it gets the job done. First is to connect to it and make it reusable – the same as a TCP socket. Note that we pass SOCK_DGRAM instead of SOCK_STREAM.

```
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
int optval = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEPORT, &optval,
    sizeof(optval));
```

Then, we can copy over our hostent struct into the sockaddr_in struct. Full definitions are provided in the man pages so it is safe to copy them over.

```
memcpy(&addr.sin_addr.s_addr, serv->h_addr, serv->h_length);
```

Then a final useful part of UDP is that we can time out receiving a packet as opposed to TCP because UDP isn't connection-oriented. The snippet to do that is below.

```
struct timeval tv;
tv.tv_sec = 0;
tv.tv_usec = SOCKET_TIMEOUT;
setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
```

Now, the socket is connected and ready to use. We can use sendto to send a packet. We should also check the return value. Note that we won't get an error if the packet isn't delivered because that is a part of the UDP protocol. We will, however, get error codes for invalid structs, bad addresses, etc.

```
char *to_send = "Hello!"
int send_ret = sendto(sock_fd, // Socket
   to_send, // Data
   strlen(to_send), // Length of data
   0, // Flags
   (struct sockaddr *)&ipaddr, // Address
   sizeof(ipaddr)); // How long the address is
```

The above code simply sends "Hello" through a UDP. There is no idea of if the packet arrives, is processed, etc.

UDP Server

There are a variety of function calls available to send UDP sockets. We will use the newer getaddrinfo to help set up a socket structure. Remember that UDP is a simple packet-based ('datagram') protocol. There is no connection to set up between the two hosts. First, initialize the hints addrinfo struct to request an IPv6, passive datagram socket.

```
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET6;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE;
```

Next, use getaddrinfo to specify the port number. We don't need to specify a host as we are creating a server socket, not sending a packet to a remote host. Be careful not to send "localhost" or any other synonym for the loop-back address. We may end up trying to passively listen to ourselves and resulting in bind errors.

```
getaddrinfo(NULL, "300", &hints, &res);

sockfd = socket(res->ai_family, res->ai_socktype,
    res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
```

The port number is less than 1024, so the program will need root privileges. We could have also specified a service name instead of a numeric port value.

So far, the calls have been similar to a TCP server. For a stream-based service, we would call listen and accept. For our UDP-server, the program can start waiting for the arrival of a packet.

```
struct sockaddr_storage addr;
int addrlen = sizeof(addr);
```

```
// ssize_t recvfrom(int socket, void* buffer, size_t buflen, int
    flags, struct sockaddr *addr, socklen_t * address_len);

byte_count = recvfrom(sockfd, buf, sizeof(buf), 0, &addr,
    &addrlen);
```

The addr struct will hold the sender (source) information about the arriving packet. Note the sockaddr_storage type is sufficiently large enough to hold all possible types of socket addresses – IPv4, IPv6 or any other Internet Protocol. The full UDP server code is below.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(int argc, char **argv) {
  struct addrinfo hints, *res;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_INET6; // INET for IPv4
  hints.ai_socktype = SOCK_DGRAM;
  hints.ai_flags = AI_PASSIVE;

  getaddrinfo(NULL, "300", &hints, &res);

  int sockfd = socket(res->ai_family, res->ai_socktype,
      res->ai_protocol);

  if (bind(sockfd, res->ai_addr, res->ai_addrlen) != 0) {
    perror("bind()");
    exit(1);
  }
  struct sockaddr_storage addr;
  int addrlen = sizeof(addr);

  while(1){
    char buf[1024];
    ssize_t byte_count = recvfrom(sockfd, buf, sizeof(buf), 0,
        &addr, &addrlen);
    buf[byte_count] = '\0';

    printf("Read %d chars\n", byte_count);
    printf("===\n");
    printf("%s\n", buf);
  }

  return 0;
}
```

Note that if you perform a partial read from a packet, the rest of that data is discarded. One call to recvfrom is one packet. To make sure that you have enough space, use 64 KiB as storage space.

Layer 7: HTTP

Layer 7 of the OSI layer deals with application-level interfaces. Meaning that you can ignore everything below this layer and treat the Internet as a way of communicating with another computer than can be secure and the session may reconnect. Common layer 7 protocols are the following

1. HTTP(S) - Hypertext Transfer Protocol. Sends arbitrary data and executes remote actions on a web server. The S standards for secure where the TCP connection uses the TLS protocol to ensure that the communication can't be read easily by an onlooker.
2. FTP - File Transfer Protocol. Transfers a file from one computer to another
3. TFTP - Trivial File Transfer Protocol. Same as above but using UDP.
4. DNS - Domain Name Service. Translates hostnames to IP addresses
5. SMTP - Simple Mail Transfer Protocol. Allows one to send plain text emails to an email server
6. SSH - Secure SHell. Allows one computer to connect to another computer and execute commands remotely.
7. Bitcoin - Decentralized cryptocurrency
8. BitTorrent - Peer to peer file sharing protocol
9. NTP - Network Time Protocol. This protocol helps keep your computer's clock synced with the outside world

What's my name?

Remember when we were talking before about converting a website to an IP address? A system called "DNS" (Domain Name Service) is used. If the IP address is missing form a machine's cache then it sends a UDP packet to a local DNS server. This server may query other upstream DNS servers.

DNS by itself is fast but insecure. DNS requests are unencrypted and susceptible to 'man-in-the-middle' attacks. For example, a coffee shop internet connection could easily subvert your DNS requests and send back different IP addresses for a particular domain. The way this is usually subverted is that after the IP address is obtained then a connection is usually made over HTTPS. HTTPS uses what is called the TLS (formerly known as SSL) to secure transmissions and verify that the hostname is recognized by a Certificate Authority. Certificate Authorities often get hacked so be careful of equating a green lock to secure. Even with this added layer of security, the united states government has recently issued a request for everyone to upgrade their DNS to DNSSec which includes additional security-focused technologies to verify with high probability that an IP address is truly associated with a hostname.

Digression aside, DNS works like this in a nutshell

1. Send a UDP packet to your DNS server
2. If that DNS server has the packet cached return the result
3. If not, ask higher-level DNS servers for the answer. Cache and send the result
4. If either packet is not answered from within a guessed timeout, resend the request.

If you want the full bits and pieces, feel free to look at the Wikipedia page. In essence, there is a hierarchy of DNS servers. First, there is the dot hierarchy. This hierarchy first resolves top-level domains `.edu` `.gov` etc. Next, it resolves the next level i.e. `illinois.edu`. Then the local resolvers can resolve any number of URLs. For example, the Illinois DNS server handles both `cs.illinois.edu` and `cs241.cs.illinois.edu`. There is a limit on how many subdomains you can have, but this is often used to route requests to different servers to avoid having to buy many high performant servers to route requests.

Non-Blocking IO

When you call `read()` if the data is unavailable, it will wait until the data is ready before the function returns. When you're reading data from a disk, that delay is short, but when you're reading from a slow network connection, requests take a long time. And the data may never arrive, leading to an unexpected close.

POSIX lets you set a flag on a file descriptor such that any call to `read()` on that file descriptor will return immediately, whether it has finished or not. With your file descriptor in this mode, your call to `read()` will start the read operation, and while it's working you can do other useful work. This is called "non-blocking" mode since the call to `read()` doesn't block.

To set a file descriptor to be non-blocking.

```
// fd is my file descriptor
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

For a socket, you can create it in non-blocking mode by adding `SOCK_NONBLOCK` to the second argument to `socket()`:

```
fd = socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, 0);
```

When a file is in non-blocking mode and you call `read()`, it will return immediately with whatever bytes are available. Say 100 bytes have arrived from the server at the other end of your socket and you call `read(fd, buf, 150)`. 'read' will return immediately with a value of 100, meaning it read 100 of the 150 bytes you asked for. Say you tried to read the remaining data with a call to `read(fd, buf+100, 50)`, but the last 50 bytes still hadn't arrived yet. `read()` would return -1 and set the global error variable **errno** to either `EAGAIN` or `EWOULDBLOCK`. That's the system's way of telling you the data isn't ready yet.

`write()` also works in non-blocking mode. Say you want to send 40,000 bytes to a remote server using a socket. The system can only send so many bytes at a time. In non-blocking mode, `write(fd, buf, 40000)` would return the number of bytes it was able to send immediately, or about 23,000. If you called `write()` right away again, it would return -1 and set errno to `EAGAIN` or `EWOULDBLOCK`. That's the system's way of telling you that it's still busy sending the last chunk of data and isn't ready to send more yet.

There are a few ways to check that your IO has arrived. Let's see how to do it using *select* and *epoll*. The first interface we have is select. It isn't preferred by many in the POSIX community if they have an alternative to it, and in most cases there is an alternative to it.

```
int select(int nfds,
fd_set *readfds,
fd_set *writefds,
fd_set *exceptfds,
struct timeval *timeout);
```

Given three sets of file descriptors, `select()` will wait for any of those file descriptors to become 'ready'.

1. `readfds` - a file descriptor in `readfds` is ready when there is data that can be read or EOF has been reached.

2. `writefds` - a file descriptor in `writefds` is ready when a call to write() will succeed.

3. `exceptfds` - system-specific, not well-defined. Just pass NULL for this.

`select()` returns the total number of ready file descriptors. If none of them become ready during the time defined by *timeout*, it will return 0. After `select()` returns, the caller will need to loop through the file descriptors in readfds and/or writefds to see which ones are ready. As readfds and writefds act as both input and output parameters, when `select()` indicates that there are ready file descriptors, it would have overwritten them to reflect only the ready file descriptors. Unless the caller intends to call `select()` only once, it would be a good idea to save a copy of readfds and writefds before calling it. Here is a comprehensive snippet.

```
fd_set readfds, writefds;
FD_ZERO(&readfds);
FD_ZERO(&writefds);
for (int i=0; i < read_fd_count; i++)
FD_SET(my_read_fds[i], &readfds);
for (int i=0; i < write_fd_count; i++)
FD_SET(my_write_fds[i], &writefds);

struct timeval timeout;
timeout.tv_sec = 3;
timeout.tv_usec = 0;
```

```
int num_ready = select(FD_SETSIZE, &readfds, &writefds, NULL,
    &timeout);

if (num_ready < 0) {
  perror("error in select()");
} else if (num_ready == 0) {
  printf("timeout\n");
} else {
  for (int i=0; i < read_fd_count; i++)
  if (FD_ISSET(my_read_fds[i], &readfds))
  printf("fd %d is ready for reading\n", my_read_fds[i]);
  for (int i=0; i < write_fd_count; i++)
  if (FD_ISSET(my_write_fds[i], &writefds))
  printf("fd %d is ready for writing\n", my_write_fds[i]);
}
```

For more information on select() The problem with select and why a lot of users don't use this or poll is that select must linearly go through each of the objects. If at any point in going through the objects, the previous objects change state, select must restart. This is highly inefficient if we have a large number of file descriptors in each of our sets. There is an alternative, that isn't much better.

epoll

epoll is not part of POSIX, but it is supported by Linux. It is a more efficient way to wait for many file descriptors. It will tell you exactly which descriptors are ready. It even gives you a way to store a small amount of data with each descriptor, like an array index or a pointer, making it easier to access your data associated with that descriptor.

First, you must create a special file descriptor with epoll_create(). You won't read or write to this file descriptor. You'll pass it to the other epoll_xxx functions and call close() on it at the end.

```
int epfd = epoll_create(1);
```

For each file descriptor that you want to monitor with epoll, you'll need to add it to the epoll data structures using epoll_ctl() with the EPOLL_CTL_ADD option. You can add any number of file descriptors to it.

```
struct epoll_event event;
event.events = EPOLLOUT; // EPOLLIN==read, EPOLLOUT==write
event.data.ptr = mypointer;
epoll_ctl(epfd, EPOLL_CTL_ADD, mypointer->fd, &event)
```

To wait for some of the file descriptors to become ready, use epoll_wait(). The epoll_event struct that it fills out will contain the data you provided in event.data when you added this file descriptor. This makes it easy for you to look up your data associated with this file descriptor.

```c
int num_ready = epoll_wait(epfd, &event, 1, timeout_milliseconds);
if (num_ready > 0) {
  MyData *mypointer = (MyData*) event.data.ptr;
  printf("ready to write on %d\n", mypointer->fd);
}
```

Say you were waiting to write data to a file descriptor, but now you want to wait to read data from it. Just use `epoll_ctl()` with the `EPOLL_CTL_MOD` option to change the type of operation you're monitoring.

```c
event.events = EPOLLOUT;
event.data.ptr = mypointer;
epoll_ctl(epfd, EPOLL_CTL_MOD, mypointer->fd, &event);
```

To unsubscribe one file descriptor from epoll while leaving others active, use `epoll_ctl()` with the `EPOLL_CTL_DEL` option.

```c
epoll_ctl(epfd, EPOLL_CTL_DEL, mypointer->fd, NULL);
```

To shut down an epoll instance, close its file descriptor.

```c
close(epfd);
```

Also to non-blocking `read()` and `write()`, any calls to `connect()` on a non-blocking socket will also be non-blocking. To wait for the connection to complete, use `select()` or epoll to wait for the socket to be writable. There are reasons to use epoll over select but due to interface, there are fundamental problems with doing so.

Blogpost about select being broken

Epoll Example

Let's break down the epoll code in the man page. We'll assume that we have a prepared TCP server socket `int listen_sock`. The first thing we have to do is create the epoll device.

```c
epollfd = epoll_create1(0);
if (epollfd == -1) {
  perror("epoll_create1");
  exit(EXIT_FAILURE);
}
```

The next step is to add the listen socket in level triggered mode.

```
// This file object will be ‘read‘ from (connect is technically a
    read operation)
ev.events = EPOLLIN;
ev.data.fd = listen_sock;

// Add the socket in with all the other fds. Everything is a file
    descriptor
if (epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_sock, &ev) == -1) {
  perror("epoll_ctl: listen_sock");
  exit(EXIT_FAILURE);
}
```

Then in a loop, we wait and see if epoll has any events.

```
struct epoll_event ev, events[MAX_EVENTS];
nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
if (nfds == -1) {
  perror("epoll_wait");
  exit(EXIT_FAILURE);
}
```

If we get an event on a client socket, that means that the client has data ready to be read, and we perform that operation. Otherwise, we need to update our epoll structure with a new client.

```
if (events[n].data.fd == listen_sock) {
  int conn_sock = accept(listen_sock, (struct sockaddr *) &addr,
      &addrlen);
  // Must set to non-blocking
  setnonblocking(conn_sock);

  // We will read from this file, and we only want to return once
  // we have something to read from. We don't want to keep getting
  // reminded if there is still data left (edge triggered)
  ev.events = EPOLLIN | EPOLLET;
  ev.data.fd = conn_sock;
  epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_sock, &ev)
}
```

The function above is missing some error checking for brevity as well. Note that this code is performant because we added the server socket in level-triggered mode and we add each of the client file descriptors in edge-triggered. Edge triggered mode leaves more calculations on the part of the application – the application must keep reading or writing until the file descriptor is out of bytes – but it prevents starvation. A more efficient implementation would also add the listening socket in edge-triggered to clear out the backlog of connections as well.

Please read through most of `man 7 epoll` before starting to program. There are a lot of gotchas. Some of the more common ones will be detailed below.

Assorted Epoll Gotchas

There are several problems with using epoll. Here we will detail a few.

1. There are two modes. Level triggered and edge-triggered. Level triggered means that while the file descriptor has events on it, it will be returned by epoll when calling the ctl function. In edge-triggered, the caller will only get the file descriptor once it goes from zero events to an event. This means if you forget to read, write, accept etc on the file descriptor until you get a EWOULDBLOCK, that file descriptor will be dropped.

2. If at any point you duplicate a file descriptor and add it to epoll, you will get an event from that file descriptor and the duplicated one.

3. You can add an epoll object to epoll. Edge triggered and level-triggered modes are the same because ctl will reset the state to zero events

4. Depending on the conditions, you may get a file descriptor that was closed from Epoll. This isn't a bug. The reason that this happens is epoll works on the kernel object level, not the file descriptor level. If the kernel object lives longer and the right flags are set, a process could get a closed file descriptor. This also means that if you close the file descriptor, there is no way to remove the kernel object.

5. Epoll has the `EPOLLONESHOT` flag which will remove a file descriptor after it has been returned in `epoll_wait`

6. Epoll using level-triggered mode could starve certain file descriptors because it is unknown how much data the application will read from each descriptor.

Read more at `man 7 epoll`

Extra: kqueue

When it comes to Event-Driven IO, the name of the game is to be fast. One extra system call is considered slow. OpenBSD and FreeBSD have an arguably better model of asynchronous IO from the kqueue model. Kqueue is a system call that is exclusive the BSDs and MacOs. It allows you to modify file descriptor events and read file descriptors all in a single call under a unified interface. So what are the benefits?

1. No more differentiation between file descriptors and kernel objects. In the epoll section, we had to discuss this distinction otherwise you may wonder why closed file descriptors are getting returned on epoll. No problem here.

2. How often do you call epoll to read file descriptors, get a server socket, and need to add another file descriptor? In a high-performance server, this can easily happen 1000s of times a second. As such, having one system call to register and grab events saves the overhead of having a system call.
3. The unified system call for all types. kqueue is the truest sense of underlying descriptor agnostic. One can add files, sockets, pipes to it and get full or near full performance. You can add the same to epoll, but Linux's whole ecosystem with async file input-output has been messed up with `aio`, meaning that since there is no unified interface, you run into weird edge cases.

Remote Procedure Calls

RPC or Remote Procedure Call is the idea that we can execute a procedure on a different machine. In practice, the procedure may execute on the same machine. However, it may be in a different context. For example, the operation under a different user with different permissions and different lifecycles.

An example of this is you may send a remote procedure call to a docker daemon to change the state of the container. Not every application needs to have access to the entire system machine, but they should have access to containers that they've created.

Privilege Separation

The remote code will execute under a different user and with different privileges from the caller. In practice, the remote call may execute with more or fewer privileges than the caller. This in principle can be used to improve the security of a system by ensuring components operate with the least privilege. Unfortunately, security concerns need to be carefully assessed to ensure that RPC mechanisms cannot be subverted to perform unwanted actions. For example, an RPC implementation may implicitly trust any connected client to perform any action, rather than a subset of actions on a subset of the data.

Stub Code and Marshaling

The stub code is the necessary code to hide the complexity of performing a remote procedure call. One of the roles of the stub code is to *marshal* the necessary data into a format that can be sent as a byte stream to a remote server.

```
// On the outside, ’getHiscore’ looks like a normal function call
// On the inside, the stub code performs all of the work to send
    and receive data to and from the remote machine.

int getHighScore(char* game) {
  // Marshal the request into a sequence of bytes:
  char* buffer;
  asprintf(&buffer,"getHiscore(%s)!", name);

  // Send down the wire (we do not send the zero byte; the ’!’
      signifies the end of the message)
  write(fd, buffer, strlen(buffer) );

  // Wait for the server to send a response
  ssize_t bytesread = read(fd, buffer, sizeof(buffer));

  // Example: unmarshal the bytes received back from text into an
      int
  buffer[bytesread] = 0; // Turn the result into a C string

  int score= atoi(buffer);
  free(buffer);
  return score;
}
```

Using a string format may be a little inefficient. A good example of this marshaling is Golang's gRPC or Google RPC. There is a version in C as well if you want to check that out.

The server stub code will receive the request, unmarshal the request into a valid in-memory data call the underlying implementation and send the result back to the caller. Often the underlying library will do this for you.

To implement RPC you need to decide and document which conventions you will use to serialize the data into a byte sequence. Even a simple integer has several common choices.

1. Signed or unsigned?
2. ASCII, Unicode Text Format 8, some other encoding?
3. Fixed number of bytes or variable depending on the magnitude.
4. Little or Big endian binary format if using binary?

To marshal a struct, decide which fields need to be serialized. It may be unnecessary to send all data items. For example, some items may be irrelevant to the specific RPC or can be re-computed by the server from the other data items present.

To marshal a linked list, it is unnecessary to send the link pointers, stream the values. As part of unmarshaling, the server can recreate a linked list structure from the byte sequence.

By starting at the head node/vertex, a simple tree can be recursively visited to create a serialized version of the data. A cyclic graph will usually require additional memory to ensure that each edge and vertex is processed exactly once.

Interface Description Language

Writing stub code by hand is painful, tedious, error-prone, difficult to maintain and difficult to reverse engineer the wire protocol from the implemented code. A better approach is to specify the data objects, messages, and services to automatically generate the client and server code. A modern example of an Interface Description Language is Google's Protocol Buffer .proto files.

Even then, Remote Procedure Calls are significantly slower (10x to 100x) and more complex than local calls. An RPC must marshal data into a wire-compatible format. This may require multiple passes through the data structure, temporary memory allocation, and transformation of the data representation.

Robust RPC stub code must intelligently handle network failures and versioning. For example, a server may have to process requests from clients that are still running an early version of the stub code.

A secure RPC will need to implement additional security checks including authentication and authorization, validate data and encrypt communication between the client and host. A lot of the time, the RPC system can do this efficiently for you. Consider if you have both an RPC client and server on the same machine. Starting up a thrift or Google RPC server could validate and route the request to a local socket which wouldn't be sent over the network.

Transferring Structured Data

Let's examine three methods of transferring data using 3 different formats - JSON, XML, and Google Protocol Buffers. JSON and XML are text-based protocols. Examples of JSON and XML messages are below.

```
<ticket><price
    currency=’dollar’>10</price><vendor>travelocity</vendor></ticket>
```

```
{ ’currency’:’dollar’ , ’vendor’:’travelocity’, ’price’:’10’ }
```

Google Protocol Buffers is an open-source efficient binary protocol that places a strong emphasis on high throughput with low CPU overhead and minimal memory copying. This means client and server stub code in multiple languages can be generated from the .proto specification file to marshal data to and from a binary stream.

Google Protocol Buffers reduces the versioning problem by ignoring unknown fields that are present in a message. See the introduction to Protocol Buffers for more information.

The general chain is to abstract away the actual business logic and the various marshaling code. If your application ever becomes CPU bound parsing XML, JSON or YAML, switch to protocol buffers!

Topics

- IPv4 vs IPv6
- TCP vs UDP
- Packet Loss/Connection Based
- Get address info
- DNS
- TCP client calls
- TCP server calls
- shutdown

- recvfrom
- epoll vs select
- RPC

Questions

- What is IPv4? IPv6? What are the differences between them?
- What is the TCP? The UDP? Give me the advantages and disadvantages of both of them. What is a scenario of using one over the other?
- Which protocol is connectionless and which one is connection based?
- What is DNS? What is the route that DNS takes?
- What does socket do?
- What are the calls to set up a TCP client?
- What are the calls to set up a TCP server?
- What is the difference between a socket shutdown and closing?
- When can you use `read` and `write`? How about `recvfrom` and `sendto`?
- What are some advantages to `epoll` over `select`? How about `select` over `epoll`?
- What is a remote procedure call? When should one use it versus HTTP or running code locally?
- What is marshaling/unmarshaling? Why is HTTP *not* an RPC?

Bibliography

[1] User Datagram Protocol. RFC 768, August 1980. URL `https://rfc-editor.org/rfc/rfc768.txt`.

[2] State of ipv6 deployment 2018, Jun 2018. URL `https://www.internetsociety.org/resources/2018/state-of-ipv6-deployment-2018/`.

[3] Danny Cohen. On holy wars and a plea for peace, Apr 1980. URL `https://www.ietf.org/rfc/ien/ien137.txt`.

[4] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014. URL `https://rfc-editor.org/rfc/rfc7231.txt`.

[5] J. Reynolds and J. Postel. Assigned numbers. RFC 1700, RFC Editor, October 1994.

12 Filesystems

/home is where the heart is

Bhuvy

Filesystems are important because they allow you to persist data after a computer is shut down, crashes, or has memory corruption. Back in the day, filesystems were expensive to use. Writing to the filesystem (FS) involved writing to magnetic tape and reading from that tape [1]. It was slow, heavy, and prone to errors.

Nowadays most of our files are stored on disk – though not all of them! The disk is still slower than memory by an order of magnitude at the least.

Some terminology before we begin this chapter. A **filesystem**, as we'll define more concretely later, is anything that satisfies the API of a filesystem. A filesystem is backed by a storage medium, such as a hard disk drive, solid state drive, RAM, etc. A disk is either a **hard disk drive (HDD)** which includes a spinning metallic platter and a head which can zap the platter to encode a 1 or a 0, or a `solid-state drive (SSD)` that can flip certain NAND gates on a chip or standalone drive to store a 1 or a 0. As of 2019, SSDs are an order of magnitude faster than the standard HDD. These are typical backings for a filesystem. A filesystem is implemented on top of this backing, meaning that we can either implement something like EXT, MinixFS, NTFS, FAT32, etc. on a commercially available hard disk. This filesystem tells the operating system how to organize the 1s and 0s to store file information as well as directory information, but more on that later. To avoid being pedantic, we'll say that a filesystem like EXT or NTFS implements the filesystem API directly (open, close, etc). Often, operating systems will add a layer of abstraction and require that the operating system satisfy its API instead (think imaginary functions linux_open, linux_close etc). The two benefits are that one filesystem can be implemented for multiple operating system APIs and adding a new OS filesystem call doesn't require all of the underlying file systems to change their API. For example, in the next iteration of linux if there was a new system call to create a backup of a file, the OS can implement that with the internal API rather than requiring all filesystem drivers to change their code.

The last piece of background is an important one. In this chapter, we will refer to sizes of files in the ISO-compliant KiB or Kibibyte. The *iB family is short for power of two storage. That means the following:

Table 12.1: Kibibyte Values

Prefix	Byte Value
KiB	1024B
MiB	1024 * 1024 B
GiB	10243̂ B

The standard notational prefixes mean the following:

Table 12.2: Kilobyte Values

Prefix	Byte Value
KB	1000B
MB	1000 * 1000 B
GB	1000$\hat{3}$ B

We will do this in the book and in the Networking chapter for the sake of consistency and to not confuse anyone. **Confusingly in the real world, there is a different convention.** That convention is that when a file is displayed in the operating system, **KB is the same as KiB**. When we are talking about computer networks, CDs, other storage **KB is not the same as KiB** and is the ISO / Metric Definition above. This is a historical quirk was brought by a clash between network developers and memory/hard storage developers. Hard storage and memory developers found that if a bit could take one of two states, it would be natural to call a Kilo- prefix 1024 because it was about 1000. Network developers had to deal with bits, real-time signal processing, and various other factors, so they went with the already accepted convention that Kilo- means 1000 of something [1]. What you need to know is if you see KB in the wild, that it may be 1024 based on the context. If any time in this class you see KB or any of the family refer to a filesystems question, you can safely infer that they are referring to 1024 as the base unit. Though when you are pushing production code, make sure to ask about the difference!

What is a filesystem?

You may have encountered the old UNIX adage, "everything is a file". In most UNIX systems, file operations provide an interface to abstract many different operations. Network sockets, hardware devices, and data on the disk are all represented by file-like objects. A file-like object must follow the following conventions:

1. It must present itself to the filesystem.
2. It must support common filesystem operations, such as `open`, `read`, `write`. At a minimum, it needs to be opened and closed.

A filesystem is an implementation of the file interface. In this chapter, we will be exploring the various callbacks a filesystem provides, some typical functionality and associated implementation details. In this class, we will mostly talk about filesystems that serve to allow users to access data on disk, which are integral to modern computers.

Here are some common features of a filesystem:

1. They deal with both storing local files and handle special devices that allow for safe communication between the kernel and user space.
2. They deal with failures, scalability, indexing, encryption, compression, and performance.
3. They handle the abstraction between a file that contains data and how exactly that data is stored on disk, partitioned, and protected.

Before we dive into the details of a filesystem, let's take a look at some examples. To clarify, a mount point is simply a mapping of a directory to a filesystem represented in the kernel.

1. `ext4` Usually mounted at / on Linux systems, this is the filesystem that usually provides disk access as you're used to.

2. `procfs` Usually mounted at /proc, provides information and control over processes.

3. `sysfs` Usually mounted at /sys, a more modern version of /proc that also allows control over various other hardware such as network sockets.

4. `tmpfs` Mounted at /tmp in some systems, an in-memory filesystem to hold temporary files.

5. `sshfs` This syncs files across the `ssh` protocol.

It tells you what filesystem directory-based system calls resolve to. For example, `/` is resolved by the `ext4` filesystem in our case, but `/proc/2` is resolved by the `procfs` system even though it contains `/` as a subsystem.

As you may have noticed, some filesystems provide an interface to things that aren't "files". Filesystems such as `procfs` are usually referred to as *virtual* filesystems, since they don't provide data access in the same sense as a traditional filesystem would. Technically, all filesystems in the kernel are represented by virtual filesystems, but we will differentiate *virtual* filesystems as filesystems that actually don't store anything on a hard disk.

The File API

A filesystem must provide callback functions to a variety of actions. Some of them are listed below:

- `open` Opens a file for IO

- `read` Read contents of a file

- `write` Write to a file

- `close` Close a file and free associated resources

- `chmod` Modify permissions of a file

- `ioctl` Interact with device parameters of character devices such as terminals

Not every filesystem supports all the possible callback functions. For example, many filesystems omit `ioctl` or `link`. Many filesystems aren't `seekable` meaning that they exclusively provide sequential access. A program cannot move to an arbitrary point in the file. This is analogous to `seekable streams`. In this chapter, we will not be examining each filesystem callback. If you would like to learn more about this interface, try looking at the documentation for Filesystems at the User Space Level (FUSE).

Storing data on disk

To understand how a filesystem interacts with data on disk, there are three key terms we will be using.

1. `disk block` A disk block is a portion of the disk that is reserved for storing the contents of a file or a directory.

2. `inode` An inode *is* a file or directory. This means that an inode contains metadata about the file as well as pointers to disk blocks so that the file can actually be written to or read from.

3. `superblock` A superblock contains metadata about the inodes and disk blocks. An example superblock can store how full each disk block is, which inodes are being used etc. Modern filesystems may actually contain multiple superblocks and a sort-of super-super block that keeps track of which sectors are governed by which superblocks. This tends to help with fragmentation.

It may seem overwhelming, but by the end of this chapter, we will be able to make sense of every part of the filesystem.

To reason about data on some form of storage – spinning disks, solid state drives, magnetic tape – it is common practice to first consider the medium of storage as a collection of *blocks*. A block can be thought of as a contiguous region on disk. While its size is sometimes determined by some property of the underlying hardware, it is more frequently determined based on the size of a page of memory for a given system, so that data from the disk can be cached in memory for faster access – a important feature of many filesystems.

A filesystem has a special block denoted as a *superblock* that stores metadata about the filesystem such as a journal (which logs changes to the filesystem), a table of inodes, the location of the first inode on disk, etc. The important thing about a superblock is that it is in a known location on disk. If not, your computer may fail to boot! Consider a simple ROM programmed into your motherboard. If your processor can't tell the motherboard to start reading and decipher a disk block to start the boot sequence, you are out of luck.

The inode is the most important structure for our filesystem as it represents a file. Before we explore it in-depth, let's list out the key information we need to have a usable file.

- Name
- File size
- Time created, last modified, last accessed
- Permissions
- Filepath
- Checksum
- File data

File Contents

From Wikipedia:

> *In a Unix-style file system, an index node, informally referred to as an inode, is a data structure used to represent a filesystem object, which can be various things including a file or a directory. Each inode stores the attributes and disk block location(s) of the filesystem object's data. Filesystem object attributes may include manipulation metadata (e.g. change, access, modify time), as well as owner and permission data (e.g. group-id, user-id, permissions).*

The superblock may store an array of inodes, each of which stores direct, and potentially several kinds of indirect pointers to disk blocks. Since inodes are stored in the superblock, most filesystems have a limit on how many inodes can exist. Since each inode corresponds to a file, this is also a limit on how many files that filesystem can have. Trying to overcome this problem by storing inodes in some other location greatly increases the complexity of the filesystem. Trying to reallocate space for the inode table is also infeasible since every byte following the end of the inode array would have to be shifted, a highly expensive operation. This isn't to say there aren't any solutions at all, although typically there is no need to increase the number of inodes since the number of inodes is usually sufficiently high.

Big idea: Forget names of files. The ‘inode’ is the file.

It is common to think of the file name as the ‘actual’ file. It’s not! Instead, consider the inode as the file. The inode holds the meta-information (last accessed, ownership, size) and points to the disk blocks used to hold the file contents. However, the inode does not usually store a filename. Filenames are usually only stored in directories (see below).

For example, to read the first few bytes of the file, follow the first direct block pointer to the first direct block and read the first few bytes. Writing follows the same process. If a program wants to read the entire file, keep reading direct blocks until you’ve read several bytes equal to the size of the file. If the total size of the file is less than that of the number of direct blocks multiplied by the size of a block, then unused block pointers will be undefined. Similarly, if the size of a file is not a multiple of the size of a block, data past the end of the last byte in the last block will be garbage.

What if a file is bigger than the maximum space addressable by its direct blocks? To that, we present a motto programmers take too seriously.

> “All problems in computer science can be solved by another level of indirection.” - David Wheeler

Except for the problem of too many layers of indirection.

To solve this problem, we introduce `indirect blocks`. A single indirect block is a block that stores pointers to more data blocks. Similarly, a double indirect block stores pointers to single indirect blocks, and the concept can be generalized to arbitrary levels of indirection. This is a important concept, as inodes are stored in the superblock, or some other structure in a well known location with a constant amount of space, indirection allows exponential increases in the amount of space an inode can keep track of.

As a worked example, suppose we divide the disk into 4KiB blocks and we want to address up to 2^{32} blocks. The maximum disk size is $4KiB * 2^{32} = 16TiB$ remember $2^{10} = 1024$. A disk block can store $\frac{4KiB}{4B}$ possible pointers or 1024 pointers. Four byte wide pointers are needed because we want to address 32 bits worth of blocks. Each pointer refers to a 4KiB disk block, so you can refer up to $1024 * 4KiB = 4MiB$ of data. For the same disk configuration, a double indirect block stores 1024 pointers to 1024 indirection tables. Thus a double-indirect block can refer up to $1024 * 4MiB = 4GiB$ of data. Similarly, a triple indirect block can refer up to 4TiB of data. This is three times as slow for reading between blocks, due to increased levels of indirection. The actual intra-block reading times don’t change.

Directory Implementation

A directory is a mapping of names to inode numbers. It is typically a normal file, but with some special bits set in its inode and a specific structure for its contents. POSIX provides a small set of functions to read the filename and inode number for each entry, which we will talk about in depth later in this chapter.

Let’s think about what directories looks like in the actual file system. Theoretically, they are files. The disk blocks will contain *directory entries* or *dirents*. What that means is that our disk block can look like this

```
| inode_num | name   | | ----------- | ------ |
| 2043567   | hi.txt | | ... |
```

Each directory entry could either be a fixed size, or a variable length C-string. It depends on how the particular filesystem implements it at the lower level. To see a mapping of filenames to inode numbers on a POSIX system, from a shell, use `ls` with the `-i` option

```
# ls -i
12983989 dirlist.c    12984068 sandwich.c
```

You can see later that this is a powerful abstraction. One can have a file be multiple different names in a directory, or exist in multiple directories.

UNIX Directory Conventions

In standard UNIX filesystems, the following entries are specially added on requests to read a directory.

1. `.` represents the current directory
2. `..` represents the parent directory
3. `~` is the name of the home directory usually

Counterintuitively, `...` could be the name of a file, not the grandparent directory. Only the current directory and the parent directory have special aliases involving `.` (namely , `.` and `..`). However, `...` *could* however be the name of a file or directory on disk (You can try this with `mkdir ...`). Confusingly, the shell `zsh` does interpret `...` as a handy shortcut to the grandparent directory (should it exist) while expanding shell commands.

Additional facts about name-related conventions:

1. Files that start with ’.’ (a period) on disk are conventionally considered ’hidden’ and will be omitted by programs like `ls` without additional flags (`-a`). This is not a feature of the filesystem, and programs may choose to ignore this.
2. Some files may also start with a NUL byte. These are usually *abstract UNIX sockets* and are used to prevent cluttering up the filesystem since they will be effectively hidden by any unexpecting program. They will, however, be listed by tools that detail information about sockets, so this is not a feature providing security.
3. If you want to annoy your neighbor, create a file with the terminal bell character. Every single time the file is listed (by calling ‘ls’, for example), an audible bell will be heard.

Directory API

While interacting with a file in C is typically done by using `open` to open the file and then `read` or `write` to interact with the file before calling `close` to release resources, directories have special calls such as, `opendir`, `closedir` and `readdir`. There is no function `writedir` since typically that implies creating a file or link. The program would use something like `open` or `mkdir`.

To explore these functions, let’s write a program to search the contents of a directory for a particular file. The code below has a bug, try to spot it!

```
int exists(char *directory, char *name) {
  struct dirent *dp;
  DIR *dirp = opendir(directory);
  while ((dp = readdir(dirp)) != NULL) {
    puts(dp->d_name);
    if (!strcmp(dp->d_name, name)) {
      return 1; /* Found */
    }
  }
  closedir(dirp);
  return 0; /* Not Found */
}
```

Did you find the bug? It leaks resources! If a matching filename is found then ‘closedir’ is never called as part of the early return. Any file descriptors opened and any memory allocated by opendir are never released. This means eventually the process will run out of resources and an `open` or `opendir` call will fail.

The fix is to ensure we free up resources in every possible code path.

In the above code, this means calling `closedir` before `return 1`. Forgetting to release resources is a common C programming bug because there is no support in the C language to ensure resources are always released with all code paths.

Given an open directory, after a call to `fork()`, either (XOR), the parent or the child can use `readdir()`, `rewinddir()` or `seekdir()`. If both the parent and the child use the above, the behavior is undefined.

There are two main gotchas and one consideration. The `readdir` function returns “.” (current directory) and “..” (parent directory). The other is programs need to explicity exclude subdirectories from a search, otherwise the search may take a long time.

For many applications, it’s reasonable to check the current directory first before recursively searching subdirectories. This can be achieved by storing the results in a linked list, or resetting the directory struct to restart from the beginning.

The following code attempts to list all files in a directory recursively. As an exercise, try to identify the bugs it introduces.

```
void dirlist(char *path) {
  struct dirent *dp;
  DIR *dirp = opendir(path);
  while ((dp = readdir(dirp)) != NULL) {
    char newpath[strlen(path) + strlen(dp->d_name) + 1];
    sprintf(newpath,"%s/%s", newpath, dp->d_name);
    printf("%s\n", dp->d_name);
    dirlist(newpath);
  }
}

int main(int argc, char **argv) {
  dirlist(argv[1]);
  return 0;
}
```

Did you find all 5 bugs?

```
// Check opendir result (perhaps user gave us a path that can not
    be opened as a directory
if (!dirp) {perror("Could not open directory"); return; }
```

```
// +2 as we need space for the / and the terminating 0
char newpath[strlen(path) + strlen(dp->d_name) + 2];

// Correct parameter
sprintf(newpath,"%s/%s", path, dp->d_name);

// Perform stat test (and verify) before recursing
if (0 == stat(newpath,&s) && S_ISDIR(s.st_mode)) dirlist(newpath)

// Resource leak: the directory file handle is not closed after
    the while loop
closedir(dirp);
```

One final note of caution. `readdir` is not thread-safe! You shouldn't use the re-entrant version of the function. Synchronizing the filesystem within a process is important, so use locks around `readdir`.

See the man page of readdir for more details.

Linking

Links are what force us to model a filesystem as a tree rather than a graph.

While modeling the filesystem as a tree would imply that every inode has a unique parent directory, links allow inodes to present themselves as files in multiple places, potentially with different names, thus leading to an inode having multiple parent directories. There are two kinds of links:

1. `Hard Links` A hard link is simply an entry in a directory assigning some name to an inode number that already has a different name and mapping in either the same directory or a different one. If we already have a file on a file system we can create another link to the same inode using the `'ln'` command:

   ```
   $ ln file1.txt blip.txt
   ```

   However, blip.txt *is* the same file. If we edit blip, I'm editing the same file as 'file1.txt!'. We can prove this by showing that both file names refer to the same inode.

   ```
   $ ls -i file1.txt blip.txt
   134235 file1.txt
   134235 blip.txt
   ```

   The equivalent C call is `link`

   ```
   // Function Prototype
   int link(const char *path1, const char *path2);

   link("file1.txt", "blip.txt");
   ```

For simplicity, the above examples made hard links inside the same directory. Hard links can be created anywhere inside the same filesystem.

2. Soft Links The second kind of link is called a soft link, symbolic link, or symlink. A symbolic link is different because it is a file with a special bit set and stores a path to another file. Quite simply, without the special bit, it is nothing more than a text file with a file path inside. Note when people generally talk about a link without specifying hard or soft, they are referring to a hard link.

   To create a symbolic link in the shell, use `ln -s`. To read the contents of the link as a file, use `readlink`. These are both demonstrated below.

   ```
   $ ln -s file1.txt file2.txt
   $ ls -i file1.txt blip.txt
   134235 file1.txt
   134236 file2.txt
   134235 blip.txt
   $ cat file1.txt
   file1!
   $ cat file2.txt
   file1!
   $ cat blip.txt
   file1!
   $ echo edited file2 >> file2.txt # >> is bash syntax for append to file
   $ cat file1.txt
   file1!
   edited file2
   $ cat file2.txt
   I'm file1!
   edited file2
   $ cat blip.txt
   file1!
   edited file2
   $ readlink myfile.txt
   file2.txt
   ```

   Note that `file2.txt` and `file1.txt` have different inode numbers, unlike the hard link, `blip.txt`.

   There is a C library call to create symlinks which is similar to link.

   ```
   symlink(const char *target, const char *symlink);
   ```

   Some advantages of symbolic links are

   - Can refer to files that don't exist yet
   - Unlike hard links, can refer to directories as well as regular files
   - Can refer to files (and directories) that exist outside of the current file system

> However, symlinks have a key disadvantage, they as slower than regular files and directories. When the link's contents are read, they must be interpreted as a new path to the target file, resulting in an additional call to open and read since the real file must be opened and read. Another disadvantage is that POSIX forbids hard linking directories where as soft links are allowed. The `ln` command will only allow root to do this and only if you provide the `-d` option. However, even root may not be able to perform this because most filesystems prevent it!

The integrity of the file system assumes the directory structure is an acyclic tree that is reachable from the root directory. It becomes expensive to enforce or verify this constraint if directory linking is allowed. Breaking these assumptions can leave file integrity tools unable to repair the file system. Recursive searches potentially never terminate and directories can have more than one parent but ".." can only refer to a single parent. All in all, a bad idea. Soft links are merely ignored, which is why we can use them to reference directories.

When you remove a file using `rm` or `unlink`, you are removing an inode reference from a directory. However, the inode may still be referenced from other directories. To determine if the contents of the file are still required, each inode keeps a reference count that is updated whenever a new link is created or destroyed. This count only tracks hard links, symlinks are allowed to refer to a non-existent file and thus, do not matter.

An example use of hard links is to efficiently create multiple archives of a file system at different points in time. Once the archive area has a copy of a particular file, then future archives can re-use these archive files rather than creating a duplicate file. This is called an incremental backup. Apple's "Time Machine" software does this.

Pathing

Now that we have definitions, and have talked about directories, we come across the concept of a path. A path is a sequence of directories that provide one with a "path" in the graph that is a filesystem. However, there are some nuances. It is possible to have a path called `a/b/../c/./`. Since `..` and `.` are special entries in directories, this is a valid path that actually refers to `a/c`. Most filesystem functions will allow uncompressed paths to be passed in. The C library provides a function `realpath` to compress the path or get the absolute path. To simplify by hand, remember that `..` means 'parent folder' and that `.` means 'current folder'. Below is an example that illustrates the simplification of the `a/b/../c/.` by using `cd` in a shell to navigate a filesystem.

1. `cd a` (in a)
2. `cd b` (in a/b)
3. `cd ..` (in a, because .. represents 'parent folder')
4. `cd c` (in a/c)
5. `cd .` (in a/c, because . represents 'current folder')

Thus, this path can be simplified to `a/c`.

Metadata

How can we distinguish between a regular file and a directory? For that matter, there are many other attributes that files also might contain. We distinguish a file type – different from the file extension i.e. png, svg, pdf – using fields inside the inode. How does the system know what type the file is?

This information is stored within an inode. To access it, use the stat calls. For example, to find out when my 'notes.txt' file was last accessed.

```
struct stat s;
stat("notes.txt", &s);
printf("Last accessed %s", ctime(&s.st_atime));
```

There are actually three versions of `stat`;

```
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

For example, a program can use `fstat` to learn about file metadata if it already has a file descriptor associated with that file.

```
FILE *file = fopen("notes.txt", "r");
int fd = fileno(file); /* Just for fun - extract the file
    descriptor from a C FILE struct */
struct stat s;
fstat(fd, & s);
printf("Last accessed %s", ctime(&s.st_atime));
```

`lstat` is almost the same as `stat` but handles symbolic links differently. From the `stat` man page.

> lstat() is identical to stat(), except that if pathname is a symbolic link, then it returns information about the link itself, not the file that it refers to.

The stat functions make use of `struct stat`. From the `stat` man page:

```
struct stat {
  dev_t     st_dev;       /* ID of device containing file */
  ino_t     st_ino;       /* Inode number */
  mode_t    st_mode;      /* File type and mode */
  nlink_t   st_nlink;     /* Number of hard links */
  uid_t     st_uid;       /* User ID of owner */
  gid_t     st_gid;       /* Group ID of owner */
  dev_t     st_rdev;      /* Device ID (if special file) */
  off_t     st_size;      /* Total size, in bytes */
  blksize_t st_blksize;   /* Block size for filesystem I/O */
  blkcnt_t st_blocks;     /* Number of 512B blocks allocated */
  struct timespec st_atim; /* Time of last access */
  struct timespec st_mtim; /* Time of last modification */
  struct timespec st_ctim; /* Time of last status change */
};
```

The `st_mode` field can be used to distinguish between regular files and directories. To accomplish this, use the macros, `S_ISDIR` and `S_ISREG`.

```c
struct stat s;
if (0 == stat(name, &s)) {
  printf("%s ", name);
  if (S_ISDIR( s.st_mode)) puts("is a directory");
  if (S_ISREG( s.st_mode)) puts("is a regular file");
} else {
  perror("stat failed - are you sure we can read this file's
      metadata?");
}
```

Permissions and bits

Permissions are a key part of the way UNIX systems provide security in a filesystem. You may have noticed that the `st_mode` field in `struct stat` contains more than the file type. It also contains the mode, a description detailing what a user can and can't do with a given file. There are usually three sets of permissions for any file. Permissions for the *user*, the *group* and *other* (every user falling outside the first two categories). For each of the three categories, we need to keep track of whetherthe user is allowed to read the file, write to the file, and execute the file. Since there are three categories and three permissions, permissions are usually represented as a 3-digit octal number. For each digit, the least significant byte corresponds to read privileges, the middle one to write privileges and the final byte to execute privileges. They are always presented as *User*, *Group*, *Other* (*UGO*). Below are some common examples. Here are the bit conventions:

1. `r` means that the set of people can read
2. `w` means that the set of people can write
3. `x` means that the set of people can execute

Table 12.3: Permissions Table

Octal Code	User	Group	Others
755	`rwx`	`r-x`	`r-x`
644	`rw-`	`r-`	`r-`

It is worth noting that the `rwx` bits have a slightly different meaning for directories. Write access to a directory that will allow a program to create or delete new files or directories inside. You can think about this as having write access to the directory entry (dirent) mappings. Read-access to a directory will allow a program to list a directory's contents. This is read access to the directory entry (dirent) mapping. Execute will allow a program to enter the directory using cd. Without the execute bit, it any attempt create or remove files or directories will fail since you cannot access them. You can, however, list the contents of the directory.

There are several command line utilities for interacting with a file's mode. `mknod` changes the type of the file. `chmod` takes a number and a file and changes the permission bits. However, before we can discuss chmod in detail, we must also understand the user ID (`uid`) and group id (`gid`) as well.

User ID / Group ID

Every user in a UNIX system has a user ID. This is a unique number that can identify a user. Similarly, users can be added to collections called groups, and every group also has a unique identifying number. Groups have a variety of uses on UNIX systems. They can be assigned capabilities - a way of describing the level of control a user has over a system. For example, a group you may have run into is the `sudoers` group, a set of trusted users who are allowed to use the command `sudo` to temporarily gain higher privileges. We'll talk more about how `sudo` works in this chapter. Every file, upon creation, an owner, the creator of the file. This owner's user ID (`uid`) can be found inside the `st_mode` file of a `struct stat` with a call to `stat`. Similarly, the group ID (`gid`) is set as well.

Every process can determine its `uid` and `gid` with `getuid` and `getgid`. When a process tries to open a file with a specific mode, it's `uid` and `gid` are compared with the `uid` and `gid` of the file. If the `uid`s match, then the process's request to open the file will be compared with the bits on the user field of the file's permissions. If the `gid`s match, then the process's request will be compared with the group field of the permissions. If none of the IDs match, then the other field will apply.

Reading / Changing file permissions

Before we discuss how to change permission bits, we should be able to read them. In C, the `stat` family of library calls can be used. To read permission bits from the command line, use `ls -l`. Note, the permissions will output in the format 'trwxrwxrwx'. The first character indicates the type of file type. Possible values for the first character include but aren't limited to.

1. (-) regular file
2. (d) directory
3. (c) character device file
4. (l) symbolic link
5. (p) named pipe (also called FIFO)
6. (b) block device
7. (s) socket

Alternatively, use the program `stat` which presents all the information that one could retrieve from the `stat` library call.

To change the permission bits, there is a system call, `int chmod(const char *path, mode_t mode);`. To simplify our examples, we will be using the command line utility of the same name `chmod` short of "change mode". There are two common ways to use `chmod`, with either an octal value or with a symbolic string.

```
$ chmod 644 file1
$ chmod 755 file2
$ chmod 700 file3
$ chmod ugo-w file4
$ chmod o-rx file4
```

The base-8 ('octal') digits describe the permissions for each role: The user who owns the file, the group and everyone else. The octal number is the sum of three values given to the three types of permission: read(4), write(2), execute(1)

Example: `chmod 755 myfile`

1. r + w + x = digit * user has 4+2+1, full permission
2. group has 4+0+1, read and execute permission
3. all users have 4+0+1, read and execute permission

Understanding the 'umask'

The umask *subtracts* (reduces) permission bits from 777 and is used when new files and new directories are created by open, mkdir etc. By default, the umask is set to 022 (octal), which means that group and other privileges will be exclusively readable. Each process has a current umask value. When forking, the child inherits the parent's umask value.

For example, by setting the umask to 077 in the shell, ensures that future file and directory creation will only be accessible to the current user,

```
$ umask 077
$ mkdir secretdir
```

As a code example, suppose a new file is created with `open()` and mode bits 666 (write and read bits for user, group and other):

```
open("myfile", O_CREAT, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
    S_IROTH | S_IWOTH);
```

If umask is octal 022, then the permissions of the created file will be 0666 & ~022 for example.

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
```

The ‘setuid’ bit

You may have noticed an additional bit that files with execute permission may have set. This bit is the `setuid` bit. It indicated that when run, the program will set the uid of the user to that of the owner of the file. Similar, there is a `setgid` bit which sets the gid of the executor to the gid of the owner. The canonical example of a program with `setuid` set is `sudo`.

`sudo` is usually a program that is owned by the root user - a user that has all capabilities. By using `sudo`, an otherwise unprivileged user can gain access to most parts of the system. This is useful for running programs that may require elevated privileges, such as using `chown` to change ownership of a file, or to use `mount` to mount or unmount filesystems (an action we will discuss later in this chapter). Here are some examples:

```
$ sudo mount /dev/sda2 /stuff/mydisk
$ sudo adduser fred
$ ls -l /usr/bin/sudo
-r-s--x--x 1 root wheel 327920 Oct 24 09:04 /usr/bin/sudo
```

When executing a process with the setuid bit, it is still possible to determine a user’s original uid with `getuid`. The real action of the `setuid` bit is to set the effective user ID (`euid`) which can be determined with `geteuid`. The actions of `getuid` and `geteuid` are described below.

- `getuid` returns the real user id (zero if logged in as root)
- `geteuid` returns the effective user id (zero if acting as root, e.g. due to the setuid flag set on a program)

These functions can allow one to write a program that can only be run by a privileged user by checking `geteuid` or go a step further and ensure that the only user who can run the code is root by using `getuid`.

The ‘sticky’ bit

Sticky bits as we use them today serve a different purpose from initial introduction. Sticky bits were a bit that could be set on an executable file that would allow a program’s text segment to remain in swap even after the end of the program’s execution. This made subsequent executions of the same program faster. Today, this behavior is no longer supported and the sticky bit only holds meaning when set on a directory,

When a directory’s sticky bit is set only the file’s owner, the directory’s owner, and the root user can rename or delete the file. This is useful when multiple users have write access to a common directory. A common use of the sticky bit is for the shared and writable `/tmp` directory where many users’ files may be stored, but users should not be able to access files belonging to other users.

To set the sticky bit, use `chmod +t`.

```
aneesh$ mkdir sticky
aneesh$ chmod +t sticky
aneesh$ ls -l
drwxr-xr-x 7 aneesh aneesh  4096 Nov 1 14:19 .
drwxr-xr-x 53 aneesh aneesh 4096 Nov 1 14:19 ..
drwxr-xr-t 2 aneesh aneesh  4096 Nov 1 14:19 sticky
```

```
aneesh$ su newuser
newuser$ rm -rf sticky
rm: cannot remove 'sticky': Permission denied
newuser$ exit
aneesh$ rm -rf sticky
aneesh$ ls -l
drwxr-xr-x 7 aneesh aneesh  4096 Nov 1 14:19 .
drwxr-xr-x 53 aneesh aneesh 4096 Nov 1 14:19 ..
```

Note that in the example above, the username is prepended to the prompt, and the command su is used to switch users.

Virtual filesystems and other filesystems

POSIX systems, such as Linux and Mac OS X (which is based on BSD) include several virtual filesystems that are mounted (available) as part of the file-system. Files inside these virtual filesystems may be generated dynamically or stored in ram. Linux provides 3 main virtual filesystems.

Table 12.4: Virtual Filesystem list

Device	Use Case
/dev	A list of physical and virtual devices (for example network card, cdrom, random number generator
/proc	A list of resources used by each process and (by tradition) set of system information
/sys	An organized list of internal kernel entities

If we want a continuous stream of 0s, we can run `cat /dev/zero`.

Another example is the file /dev/null, a great place to store bits that you never need to read. Bytes sent to /dev/null/ are never stored and simply discarded. A common use of /dev/null is to discard standard output. For example,

```
$ ls . >/dev/null
```

Managing files and filesystems

Given the multitude of operations that are available to you from the filesystem, let's explore some tools and techniques that can be used to manage files and filesystems.

One example is creating a secure directory. Suppose you created your own directory in /tmp and then set the permissions so that only you can use the directory (see below). Is this secure?

```
$ mkdir /tmp/mystuff
$ chmod 700 /tmp/mystuff
```

There is a window of opportunity between when the directory is created and when it's permissions are changed. This leads to several vulnerabilities that are based on a race condition.

Another user replaces `mystuff` with a hard link to an existing file or directory owned by the second user, then they would be able to read and control the contents of the `mystuff` directory. Oh no - our secrets are no longer secret!

However in this specific example, the `/tmp` directory has the sticky bit set, so only the owner may delete the `mystuff` directory, and the simple attack scenario described above is impossible. This does not mean that creating the directory and then later making the directory private is secure! A better version is to atomically create the directory with the correct permissions from its inception.

```
$ mkdir -m 700 /tmp/mystuff
```

Obtaining Random Data

`/dev/random` is a file that contains a random number generator where the entropy is determined from environmental noise. Random will block/wait until enough entropy is collected from the environment.

`/dev/urandom` is like random, but differs in the fact that it allows for repetition (lower entropy threshold), thus won't block.

One can think of both of these as streams of characters from which a program can read as opposed to files with a start and end. To touch on a misconception, most of the time one should be using `/dev/urandom`. The only specific use case of `/dev/random` is when one needs cryptographically secure data on bootup and the system should block. Otherwise, there are the following reasons.

1. Empirically, they both produce numbers that look random enough.
2. `/dev/random` may block at an inconvenient time. If one is programming a service for high scalability and relies on `/dev/random`, an attacker can reliably exhaust the entropy pool and cause the service to block.
3. Manual page authors pose a hypothetical attack where an attacker exhausts the entropy pool and guesses the seeding bits, but that attack has yet to be implemented.
4. Some operating system don't have a true `/dev/random` like MacOS.
5. Security experts will talk about Computational Security vs Information Theoretic security, more on this article Urandom Myths. Most encryption is computationally secure, which means `/dev/urandom` is as well.

Copying Files

Use the versatile `dd` command. For example, the following command copies 1 MiB of data from the file `/dev/urandom` to the file `/dev/null`. The data is copied as 1024 blocks of block size 1024 bytes.

```
$ dd if=/dev/urandom of=/dev/null bs=1k count=1024
```

Both the input and output files in the example above are virtual - they don't exist on a disk. This means the speed of the transfer is unaffected by hardware power.

dd is also commonly used to make a copy of a disk or an entire filesystem to create images that can either be burned on to other disks or to distribute data to other users.

Updating Modification Time

The touch executable creates a file if it is non-existant and also updates the file's last modified time to be the current time. For example, we can make a new private file with the current time:

```
$ umask 077     # all future new files will mask out all
    r,w,x bits for group and other access
$ touch file123 # create a file if it non-existant, and
    update its modified time
$ stat file123
  File: ‘file123’
  Size: 0         Blocks: 0        IO Block: 65536 regular
      empty file
Device: 21h/33d Inode: 226148   Links: 1
Access: (0600/-rw-------) Uid: (395606/ angrave) Gid:
    (61019/ ews)
Access: 2014-11-12 13:42:06.000000000 -0600
Modify: 2014-11-12 13:42:06.001787000 -0600
Change: 2014-11-12 13:42:06.001787000 -0600
```

An example use of touch is to force make to recompile a file that is unchanged after modifying the compiler options inside the makefile. Remember that make is 'lazy' - it will compare the modified time of the source file with the corresponding output file to see if the file needs to be recompiled.

```
$ touch myprogram.c # force my source file to be recompiled
$ make
```

Managing Filesystems

To manage filesystems on your machine, use mount. Using mount without any options generates a list (one filesystem per line) of mounted filesystems including networked, virtual and local (spinning disk / SSD-based) filesystems. Here is a typical output of mount

```
$ mount
/dev/mapper/cs241--server_sys-root on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
tmpfs on /dev/shm type tmpfs
    (rw,rootcontext="system_u:object_r:tmpfs_t:s0")
/dev/sda1 on /boot type ext3 (rw)
/dev/mapper/cs241--server_sys-srv on /srv type ext4 (rw)
/dev/mapper/cs241--server_sys-tmp on /tmp type ext4 (rw)
/dev/mapper/cs241--server_sys-var on /var type ext4
    (rw)rw,bind)
/srv/software/Mathematica-8.0 on /software/Mathematica-8.0
    type none (rw,bind)
engr-ews-homes.engr.illinois.edu:/fs1-homes/angrave/linux
    on /home/angrave type nfs
    (rw,soft,intr,tcp,noacl,acregmin=30,vers=3,sec=sys,sloppy,addr=128.174.252.10
```

Notice that each line includes the filesystem type source of the filesystem and mount point. To reduce this output, we can pipe it into grep and only see lines that match a regular expression.

```
>mount | grep proc # only see lines that contain 'proc'
proc on /proc type proc (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
```

Filesystem Mounting

Suppose you had downloaded a bootable Linux disk image from the arch linux download page

```
$ wget $URL
```

Before putting the filesystem on a CD, we can mount the file as a filesystem and explore its contents. Note: mount requires root access, so let's run it using sudo

```
$ mkdir arch
$ sudo mount -o loop archlinux-2015.04.01-dual.iso ./arch
$ cd arch
```

Before the mount command, the arch directory is new and obviously empty. After mounting, the contents of arch/ will be drawn from the files and directories stored in the filesystem stored inside the archlinux-2014.11.01-dual.iso file. The loop option is required because we want to mount a regular file, not a block device such as a physical disk.

The loop option wraps the original file as a block device. In this example, we will find out below that the file system is provided under /dev/loop0. We can check the filesystem type and mount options by running the mount command without any parameters. We will pipe the output into grep so that we only see the relevant output line(s) that contain 'arch'.

```
$ mount | grep arch
/home/demo/archlinux-2014.11.01-dual.iso on /home/demo/arch type
    iso9660 (rw,loop=/dev/loop0)
```

The iso9660 filesystem is a read-only filesystem originally designed for optical storage media (i.e. CDRoms). Attempting to change the contents of the filesystem will fail

```
$ touch arch/nocando
touch: cannot touch '/home/demo/arch/nocando': Read-only file
    system
```

Memory Mapped IO

While we traditionally think of reading and writing from a file as an operation that happens by using the read and write calls, there is an alternative, mapping a file into memory using mmap. mmap can also be used for IPC, and you can see more about mmap as a system call that enables shared memory in the IPC chapter. In this chapter, we'll briefly explore mmap as a filesystem operation.

mmap takes a file and maps its contents into memory. This allows a user to treat the entire file as a buffer in memory for easier semantics while programming, and to avoid having to read a file as discrete chunks explicitly.

Not all filesystems support using mmap for IO. Those that do have varying behavior. Some will simply implement mmap as a wrapper around read and write. Others will add additional optimizations by taking advantage of the kernel's page cache. Of course, such optimization can be used in the implementation of read and write as well, so often using mmap has identical performance.

mmap is used to perform some operations such as loading libraries and processes into memory. If many programs only need read-access to the same file, then the same physical memory can be shared between multiple processes. This is used for common libraries like the C standard library.

The process to map a file into memory is as follows.

1. mmap requires a file descriptor, so we need to open the file first
2. We seek to our desired size and write one byte to ensure that the file is sufficient length
3. When finished call munmap to unmap the file from memory.

Here is a quick example.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>


int fail(char *filename, int linenumber) {
  fprintf(stderr, "%s:%d %s\n", filename, linenumber,
      strerror(errno));
  exit(1);
  return 0; /*Make compiler happy */
}
#define QUIT fail(__FILE__, __LINE__ )

int main() {
  // We want a file big enough to hold 10 integers
  int size = sizeof(int) * 10;

  int fd = open("data", O_RDWR | O_CREAT | O_TRUNC, 0600); //6 =
      read+write for me!

  lseek(fd, size, SEEK_SET);
  write(fd, "A", 1);

  void *addr = mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED,
      fd, 0);
  printf("Mapped at %p\n", addr);
  if (addr == (void*) -1 ) QUIT;

  int *array = addr;
  array[0] = 0x12345678;
  array[1] = 0xdeadc0de;

  munmap(addr,size);
  return 0;

}
```

The careful reader may notice that our integers were written in least-significant-byte format because that is the endianness of the CPU that we ran this example on. We also allocated a file that is one byte too many! The `PROT_READ | PROT_WRITE` options specify the virtual memory protection. The option `PROT_EXEC` (not used here) can be set to allow CPU execution of instructions in memory.

Reliable Single Disk Filesystems

Most filesystems cache significant amounts of disk data in physical memory. Linux, in this respect, is extreme. All unused memory is used as a giant disk cache. The disk cache can have a significant impact on overall system performance because disk I/O is slow. This is especially true for random access requests on spinning disks where the disk read-write latency is dominated by the seek time required to move the read-write disk head to the correct position.

For efficiency, the kernel caches recently used disk blocks. For writing, we have to choose a trade-off between performance and reliability. Disk writes can also be cached ("Write-back cache") where modified disk blocks are stored in memory until evicted. Alternatively, a 'write-through cache' policy can be employed where disk writes are sent immediately to the disk. The latter is safer as filesystem modifications are quickly stored to persistent media but slower than a write-back cache. If writes are cached then they can be delayed and efficiently scheduled based on the physical position of each disk block. Note, this is a simplified description because solid state drives (SSDs) can be used as a secondary write-back cache.

Both solid state disks (SSD) and spinning disks have improved performance when reading or writing sequential data. Thus, operating systems can often use a read-ahead strategy to amortize the read-request costs and request several contiguous disk blocks per request. By issuing an I/O request for the next disk block before the user application requires the next disk block, the apparent disk I/O latency can be reduced.

If your data is important and needs to be force written to disk, call `sync` to request that a filesystem's changes be written (flushed) to disk. However, operating systems may ignore this request. Even if the data is evicted from the kernel buffers, the disk firmware may use an internal on-disk cache or may not yet have finished changing the physical media. Note, you can also request that all changes associated with a particular file descriptor are flushed to disk using `fsync(int fd)`. There is a fiery debate about this call being useless, initiated by PostgresQL's team https://lwn.net/Articles/752063/

If your operating system fails in the middle of an operation, most modern file systems do something called **journaling** to work around this. What the file system does is before it completes a potentially expensive operation, is that it writes what it is going to do down in a journal. In the case of a crash or failure, one can step through the journal and see which files are corrupt and fix them. This is a way to salvage hard disks in cases there is critical data and there is no apparent backup.

Even though it is unlikely for your computer, programming for data centers means that disks fail every few seconds. Disk failures are measured using "Mean-Time-To-Failure (MTTF)". For large arrays, the mean failure time can be surprisingly short. If the MTTF(single disk) = 30,000 hours, then the MTTF(1000 disks)= 30000/1000=30 hours or about a day and a half! That's also assuming that the failures between the disks are independent, which they often aren't.

RAID - Redundant Array of Inexpensive Disks

One way to protect against this is to store the data twice! This is the main principle of a "RAID-1" disk array. By duplicating the writes to a disk with writes to another backup disk, there are exactly two copies of the data. If one disk fails, the other disk serves as the only copy until it can be re-cloned. Reading data is faster since data can be requested from either disk, but writes are potentially twice as slow because now two write commands need to be issued for every disk block write. Compared to using a single disk, the cost of storage per byte has doubled.

Another common RAID scheme is RAID-0, meaning that a file could be split up among two disks, but if any disk fails then the files are irrecoverable. This has the benefit of halving write times because one part of the file could be writing to hard disk one and another part to hard disk two.

It is also common to combine these systems. If you have a lot of hard disks, consider RAID-10. This is where you have two systems of RAID-1, but the systems are hooked up in RAID-0 to each other. This means you would get roughly the same speed from the slowdowns but now any one disk can fail and you can recover that disk. If two disks from opposing raid partitions fail, there is a chance that you can recover though we don't could on it most of the time.

Higher Levels of RAID

RAID-3 uses parity codes instead of mirroring the data. For each N-bits written, we will write one extra bit, the 'Parity bit' that ensures the total number of 1s written is even. The parity bit is written to an additional disk. If any disk including the parity disk is lost, then its contents can still be computed using the contents of the other disks.

One disadvantage of RAID-3 is that whenever a disk block is written, the parity block will always be written too. This means that there is effectively a bottleneck in a separate disk. In practice, this is more likely to cause a failure because one disk is being used 100% of the time and once that disk fails then the other disks are more prone to failure.

A single disk failure is recoverable because there is sufficient data to rebuild the array from the remaining disks. Data-loss will occur when two disks are unusable because there is no longer sufficient data to rebuild the array. We can calculate the probability of a two disk failure based on the repair time which factors both the time to insert a new disk and the time required to rebuild the entire contents of the array.

```
MTTF = mean time to failure
MTTR = mean time to repair
N = number of original disks

p = MTTR / (MTTF-one-disk / (N-1))
```

Using typical numbers (MTTR=1day, MTTF=1000days, N-1 = 9, p=0.009)

There is a 1% chance that another drive will fail during the rebuild process (at that point you had better hope you still have an accessible backup of your original data. In practice, the probability of a second failure during the repair process is likely higher because rebuilding the array is I/O-intensive (and on top of normal I/O request activity). This higher I/O load will also stress the disk array.

RAID-5 is similar to RAID-3 except that the check block (parity information) is assigned to different disks for different blocks. The check-block is 'rotated' through the disk array. RAID-5 provides better read and write performance than RAID-3 because there is no longer the bottleneck of the single parity disk. The one drawback is that you need more disks to have this setup, and there are more complicated algorithms that need to be used.

Failure is common. Google reports 2-10% of disks fail per year. Multiplying that by 60,000+ disks in a single warehouse. Services must survive single disk, rack of servers, or whole data center failures.

Solutions

Simple redundancy (2 or 3 copies of each file) e.g., Google GFS (2001). More efficient redundancy (analogous to RAID 3++) e.g., Google Colossus filesystem (~2010): customizable replication including Reed-Solomon codes with 1.5x redundancy

Simple Filesystem Model

Software developers need to implement filesystems all the time. If that is surprising to you, we encourage you to take a look at Hadoop, GlusterFS, Qumulo, etc. Filesystems are hot areas of research as of 2018 because people have realized that the software models that we have devised don't take full advantage of our current hardware. Additionally, the hardware that we use for storing information is getting better all the time. As such, you may end up designing a filesystem yourself someday. In this section, we will go over one of a fake filesystems and "walk through" some examples of how things work.

So, what does our hypothetical filesystem look like? We will base it off of the `minixfs`, a simple filesystem that happens to be the first filesystem that Linux ran on. It is laid out sequentially on disk, and the first section is the superblock. The superblock stores important metadata about the entire filesystem. Since we want to be able to read this block before we know anything else about the data on disk, this needs to be in a well-known location so the start of the disk is a good choice. After the superblock, we'll keep a map of which inodes are being used. The nth bit is set if the nth inode – 0 being the inode root – is being used. Similarly, we store a map recording which data blocks are used. Finally, we have an array of inodes followed by the rest of the disk - implicitly partitioned into data blocks. One data block may be identical to the next from the perspective of the hardware components of the disk. Thinking about the disk as an array of data blocks is simply something we do so that we have a way to describe where files live on disk.

Below, we have an example of how an inode that describes a file may look. Note that for the sake of simplicity, we have drawn arrows mapping data block numbers in the inode to their locations on disk. These aren't pointers so much as indices into an array.

We will assume that a data block is 4 KiB.

Note that a file will fill up each of its data blocks completely before requesting an additional data block. We will refer to this property as the file being *compact*. The file presented above is interesting since it uses all of its direct blocks, one of the entries for its indirect block and partially uses another indirect block.

The following subsections will all refer to the file presented above.

File Size vs Space on Disk

Our file's size must be stored in the inode. The filesystem isn't aware of the actual contents of what is in a file - that data is considered the user's and should only be manipulated by the user. However, we can compute upper and lower bounds on the filesize by only looking at how many blocks the file uses.

There are two full direct blocks, which together store $2 * sizeof(data\_block) = 2 * 4KiB = 8KiB$.

There are two used blocks referenced by the indirect block, which can store up to $8KiB$ as calculated above.

We can now add these values to get an upper bound on the file size of $16KiB$.

What about a lower bound? We know that we must use the two direct blocks, one block referenced by the indirect block and at least 1 byte of a second block referenced by the indirect block. With this information, we can work out the lower bound to be $2 * 4KiB + 4KiB + 1 = 12KiB + 1B$.

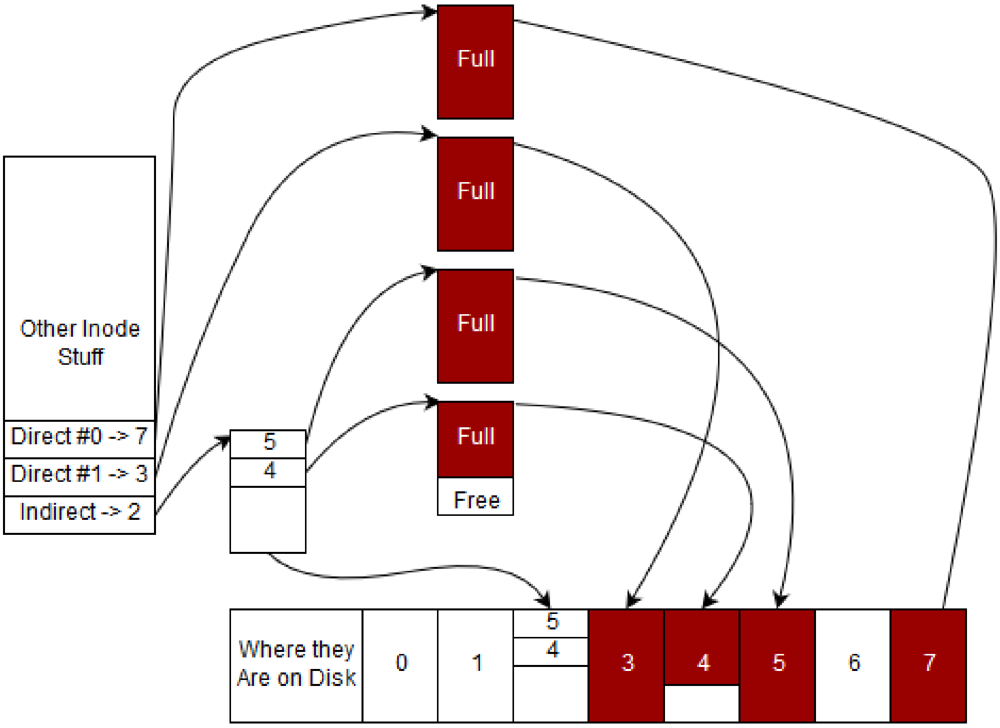


Figure 12.1: Sample file filling up

Note that our calculations so far have been to determine how much data the user is storing on disk. What about the *overhead* of storing this data incurred while using this filesystem? You'll notice that we use an indirect block to store the disk block numbers of blocks used beyond the two direct blocks. While doing our above calculations, we omitted this block. This would instead be counted as the overhead of the file, and thus the total overhead of storing this file on disk is $sizeof(indirect\_block) = 4KiB$).

Thinking about overhead, a related calculation could be to determine the max/min disk usage per file in this filesystem.

Trivially a file of size 0 has no associated data blocks and takes up no space on disk (ignoring the space required for the inode since these are located in a fixed size array somewhere on disk). How about the disk usage of the smallest non-empty file? That is, consider a file of size $1B$. Note that when a user writes the first byte, a data block will be allocated. Since each data block is $4KiB$, we find that $4KiB$ is the minimum disk usage for a non-empty file. Here, we observe that the file size will only be $1B$, despite that $4KiB$ of the disk is used – there is a distinction between file size and disk usage because of overhead!

Finding maximum is slightly more involved. As we saw earlier in this chapter, a filesystem with this structure can have 1024 data block numbers in one indirect block. This implies that the maximum filesize can be $2 * 4KiB + 1024 * 4KiB = 4MiB + 8KiB$ (after accounting for the direct blocks as well). However, on disk we also store the indirect block itself. This means that an additional $4KiB$ of overhead will be used to account for the indirect block, so the total disk usage will be $4MiB + 12KiB$.

Note that when only using direct blocks, completely filling up a direct block implies that our filesize and our disk usage are the same thing! While it would seem like we always want this ideal scenario, it puts a restrictive limit on the maximum filesize. Attempting to remedy this by increasing the number of direct blocks seems promising, but note that this requires increasing the size of an inode and reducing the amount of space available to store user data – a tradeoff you will have to evaluate for yourself. Alternatively always trying to split your data up into chunks that never use indirect blocks is may exhaust the limited pool of available inodes.

Performing Reads

Performing reads tend to be pretty easy in our filesystem because our files are compact. Let's say that we want to read the entirety of this particular file. What we'd start by doing is go to the inode's direct struct and find the first direct data block number. In our case, it is #7. Then we find the 7th data block from the *start* of all data blocks. Then we read all of those bytes. We do the same thing for all of the direct nodes. What do we do after? We go to the indirect block and read the indirect block. We know that every 4 bytes of the indirect block is either a sentinel node (-1) or the number of *another* data block. In our particular example, the first four bytes evaluate to the integer 5, meaning that our data continues on the 5th data block from the beginning. We do the same for data block #4 and we stop after because we exceed the size of the inode

Now, let's think about the edge cases. How would a program start the read starting at an arbitrary offset of n bytes given that block sizes are $4KiBs$. How many indirect blocks should there be if the filesystem is correct? (Hint: *think about using the size of the inode*)

Performing Writes

Writing to files

Performing writes fall into two categories, writes to files and writes to directories. First we'll focus on files and assume that we are writing a byte to the 6th KiB of our file. To perform a write on a file at a particular offset, first the filesystem must go to the data block would start at that offset. For this particular example we would have to go to the 2nd or indexed number 1 inode to perform our write. We would once again fetch this number from the inode, go to the root of the data blocks, go to the 5th data block and perform our write at the 2KiB offset from this block because we skipped the first four kibibytes of the file in block 7. We perform our write and go on our merry way.

Some questions to consider.

- How would a program perform a write go across data block boundaries?
- How would a program perform a write after adding the offset would extend the length of the file?
- How would a program perform a write where the offset is greater than the length of the original file?

Writing to directories

Performing a write to a directory implies that an inode needs to be added to a directory. If we pretend that the example above is a directory. We know that we will be adding at most one directory entry at a time. Meaning that we have to have enough space for one directory entry in our data blocks. Luckily the last data block that we have has enough free space. This means we need to find the number of the last data block as we did above, go to where the data ends, and write one directory entry. Don't forget to update the size of the directory so that the next creation doesn't overwrite your file!

Some more questions:

- How would would a program perform a write when the last data block is already full?
- How about when all the direct blocks have been filled up and the inode doesn't have an indirect block?
- What about when the first indirect entry (#4) is full?

Adding Deletes

If the inode is a file, then remove the directory entry in the parent directory by marking it as invalid (maybe making it point to inode -1) and skip it in your reads. A filesystem decreases the hard link count of the inode and if the count reaches zero, free the inode in the inode map and free all associated data blocks so they are reclaimed by the filesystem. In many operating systems, several fields in the inode get overwritten.

If the inode is a directory, the filesystem checks if it is empty. If not, then the kernel will most likely mark an error.

Extra: Modern Filesystems

While the API for most filesystems have stayed the same on POSIX over the years, the actual filesystems themselves provide lots of important aspects.

- Data Integrity. File systems use journaling and sometimes checksums to ensure that the data written to is valid. Journalling is a simple invention where the file system writes an operation in a journal. If the filesystem crashes before the operation is complete, it can resume the operation when booted up again using the partial journal.
- Caching. Linux does a good job of caching file system operations like finding inodes. This makes disk operations seem nearly instant. If you want to see a slow system, look at Windows with FAT/NTFS. Disk operations need to be cached by the application, or it will burn through the CPU.
- Speed. On spinning disk machines, data that is toward the end of a metallic platter will spin faster (angular velocity is farther from the center). Programs used this to reduce time loading large files like movies in a video editing piece of software. SSDs don't have this problem because there is no spinning disk, but they will portion off a section of their space to be used as "swap space" for fiels.
- Parallelism. Filesystems with multiple heads (for physical hard disks) or multiple controllers (for SSDs) can utilize parallelism by multiplexing the PCIe slot with data, always serving some data to the application whenever possible.
- Encryption. Data can be encrypted with one or more keys. A good example of this is Apple's APFS file systems.
- Redundancy. Sometimes data can be replicated to blocks to ensure that the data is always available.
- Efficient Backups. Many of us have data that we can't store on the cloud for one reason or another. It is useful that when a filesystems is either being used as a backup medium or is the source to the backup that it is able to calculate what has changed efficiently, compress files, and sync between the external drive.
- Integriy and Bootability. File systems need to be resillient to bit flipping. Most readers have their operating system installed on the same paritition as the file system that they used to do different operations. The file system needs to make sure a stray read or write doesn't destroy the boot sector – meaning your computer can't start up again.
- Fragmentation. Just like a memory allocator, allocating space for a file leads to both internal and external fragmentation. The same caching benefit occurs when disk blocks for a single file are located next to each other. File systems need to perform well under low, high, and possible fragmentation usage.

- Distributed. Sometimes, the filesystem should be single machine fault tolerant. Hadoop and other distributed file system allow you to do that.

Extra: Cutting Edge File systems

There are a few filesystem hardware nowadays that are truly cutting edge. The one we'd briefly like to touch on is AMD's StoreMI. We aren't trying to sell AMD chipsets, but the featureset of StoreMI warrants a mention.

StoreMI is a hardware microcontroller that analyzes how the operating system accesses files and moves files/blocks around to speed up the load time. A common usage can be imagined as having a fast, but small capacity SSD and a slower, large capcity HDD. To make it seem like all the files are on an SSD, the StoreMI matches the pattern of file access. If you are starting up Windows, Windows will often access many files in the same order. StoreMI takes note of that and when the microcontroller notices it is starting the boot, it will move files from the HDD drive to the SSD before they are requested by the operating system. By the time the operating system needs then, they are already on the SSD. StoreMI also does this with other applications as well. The technology still has a lot to be desired for, but it is an interesting intersection of data and pattern matching with filesystems.

Topics

- Superblock
- Data Block
- Inode
- Relative Path
- File Metadata
- Hard and Soft Links
- Permission Bits
- Mode bits
- Working with Directories
- Virtual File System
- Reliable File Systems
- RAID

Questions

- How big can files be on a file system with 15 Direct blocks, 2 double, 3 triple indirect, 4kb blocks and 4byte entries? (Assume enough infinite blocks)
- What is a superblock? Inode? Data block?

- How do we simplify `/./proc/../dev/./random`/
- In ext2, what is stored in an inode, and what is stored in a directory entry?
- What are /sys, /proc, /dev/random, and /dev/urandom?
- What are the permission bits?
- How does one use chmod to set user/group/owner read/write/execute permissions?
- What does the “dd” command do?
- What is the difference between a hard link and a symbolic link? Does the file need to exist?
- “ls -l” shows the size of each file in a directory. Is the size stored in the directory or in the file’s inode?

Bibliography

[1] International. URL `https://www.iec.ch/si/binary.htm`.

13
Signals

That's a signal, Jerry, that's a signal! [snaps his fingers again] Signal!
George Costanza (Seinfeld)

Signals are a convenient way to deliver low-priority information and for users to interact with their programs when other ways don't work (for example standard input being frozen). They allow a program to clean up or perform an action in the case of an event. Sometimes, a program can choose to ignore events which is supported. Crafting a program that uses signals well is tricky due to how signals are handled. As such, signals are usually for termination and clean up. Rarely are they supposed to be used in programming logic.

For those of you with an architecture background, the interrupts used here aren't the interrupts generated by the hardware. Those interrupts are almost always handled by the kernel because they require higher levels of privileges. Instead, we are talking about software interrupts that are generated by the kernel – though they can be in response to a hardware event like SIGSEGV.

This chapter will go over how to read information from a process that has either exited or been signaled. Then, it will deep dive into what are signals, how does the kernel deal with a signal, and the various ways processes can handle signals both with and without threads.

The Deep Dive of Signals

A signal allows one process to send an event or message to another process. If that process wants to accept the signal, it can, and then, for most signals, decide what to do with that signal.

First, a bit of terminology. A is a per-process attribute that determines how a signal is handled after it is **delivered**. Think of it as a table of signal-action pairs. The full discussion is in the Man Page. The actions are

1. `TERM`, terminates the process
2. `IGN`, ignore
3. `CORE`, generate a core dump
4. `STOP`, stops a process
5. `CONT`, continues a process
6. Execute a custom function.

A determines whether a particular signal is delivered or not. The overall process for how a kernel sends a signal are below.

1. If no signals have arrived, the process can install its own signal handlers. This tells the kernel that when the process gets signal X that it should jump to function Y.
2. A signal that is created is in a "generated" state.
3. The time between when a signal is generated and the kernel can apply the mask rules is called the pending state.
4. Then the kernel then checks the process' signal mask. If the mask says all the threads in a process are blocking the signal, then the signal is currently blocked and nothing happens until a thread unblocks it.
5. If a single thread can accept the signal, then the kernel executes the action in the disposition table. If the action is a default action, then no threads need to be paused.
6. Otherwise, the kernel delivers the signal by stopping *whatever* a particular thread is doing currently, and jumps that thread to the signal handler. The signal is now in the delivered phase. More signals can be generated now, but they can't be delivered until the signal handler is complete which is when the delivered phase is over.
7. Finally, we consider a signal caught if the process remains intact after the signal was delivered.

As a flowchart

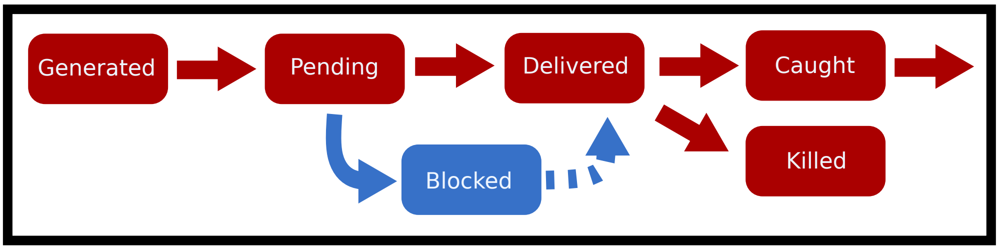


Figure 13.1: Signal lifecycle diagram

Here are some common signals that you will see thrown around.

One of our favorite anecdotes is to never use `kill -9` for a host of reasons. The following is an excerpt from Useless Use of Kill -9 Link to archive

> No no no. Don't use kill -9.
>
> It doesn't give the process a chance to cleanly:
>
> 1) shut down socket connections
>
> 2) clean up temp files
>
> 3) inform its children that it is going away
>
> 4) reset its terminal characteristics
>
> and so on and so on and so on.
>
> Generally, send 15, and wait a second or two, and if that doesn't work, send 2, and if that doesn't work, send 1. If that doesn't, REMOVE THE BINARY because the program is badly behaved!
>
> Don't use kill -9. Don't bring out the combine harvester just to tidy up the flower pot.

Table 13.1: POSIX Signals

Name Usual Use	Portable Number	Default Action
SIGINT Stop a process nicely	2	Terminate (Can be caught)
SIGQUIT Stop a process harshly	3	Terminate (Can be caught)
SIGTERM Stop a process even more harshly	15	Terminate Process
SIGSTOP Suspends a process	N/A	Stop Process (Cannot be caught)
SIGCONT Starts after a stop	N/A	Continues a process
SIGKILL You want the process gone	9	Terminate Process (Cannot be caught)

We still keep kill -9 in there for extreme scenarios where the process needs to be gone.

Sending Signals

Signals can be generated in multiple ways.

1. The user can send a signal. For example, you are at the terminal, and you press CTRL-C. One can also use the built-in kill to send any signal.

2. The system can send an event. For example, if a process accesses a page that it isn't supposed to, the hardware generates an interrupt which gets intercepted by the kernel. The kernel finds the process that caused this and sends a signal SIGSEGV. There are other kernel events like a child being created or a process needs to be resumed.

3. Finally, another process can send a message. This could be used in low-stakes communication of events between processes. If you are relying on signals to be the driver in your program, you should rethink your application design. There are many drawbacks to using POSIX/Real-Time signals for asynchronous communication. The best way to handle interprocess communication is to use, well, interprocess communication methods specifically designed for your task at hand.

You or another process can temporarily pause a running process by sending it a SIGSTOP signal. If it succeeds, it will freeze a process. The process will not be allocated any more CPU time. To allow a process to resume execution, send it the SIGCONT signal. For example, the following is a program that slowly prints a dot every second, up to 59 dots.

```
#include <unistd.h>
#include <stdio.h>
int main() {
  printf("My pid is %d\n", getpid() );
  int i = 60;
  while(--i) {
    write(1, ".",1);
    sleep(1);
  }
  write(1, "Done!",5);
  return 0;
}
```

We will first start the process in the background (notice the & at the end). Then, send it a signal from the shell process by using the kill command.

```
$ ./program &
My pid is 403
...
$ kill -SIGSTOP 403
$ kill -SIGCONT 403
...
```

In C, a program can send a signal to the child using kill POSIX call,

```
kill(child, SIGUSR1); // Send a user-defined signal
kill(child, SIGSTOP); // Stop the child process (the child cannot
    prevent this)
kill(child, SIGTERM); // Terminate the child process (the child
    can prevent this)
kill(child, SIGINT); // The equivalent to CTRL-C (by default
    closes the process)
```

As we saw above there is also a kill command available in the shell. Another command killall works the exact same way but instead of looking up by , it tries to match the name of the process. ps is an important utility that can help you find the pid of a process.

```
# First let's use ps and grep to find the process we want to send
    a signal to
$ ps au | grep myprogram
angrave 4409   0.0  0.0  2434892   512 s004 R+    2:42PM   0:00.00
    myprogram 1 2 3

#Send SIGINT signal to process 4409 (The equivalent of `CTRL-C`)
$ kill -SIGINT 4409

# Send SIGKILL (terminate the process)
$ kill -SIGKILL 4409
$ kill -9 4409
```

```
# Use kill all instead to kill a process by executable name
$ killall -l firefox
```

To send a signal to the running process, use raise or kill with getpid().

```
raise(int sig); // Send a signal to myself!
kill(getpid(), int sig); // Same as above
```

For non-root processes, signals can only be sent to processes of the same user. You can't SIGKILL any process! man -s2 kill for more details.

Handling Signals

There are strict limitations on the executable code inside a . Most library and system calls are async-signal-unsafe, meaning they may not be used inside a signal handler because they are not re-entrant. Re-entrant safety means that your function can be frozen at any point and executed again, can you guarantee that your function wouldn't fail? Let's take the following

```
int func(const char *str) {
  static char buffer[200];
  strncpy(buffer, str, 199);
  # Here is where we get paused
  printf("%s\n", buffer)
}
```

1. We execute (func("Hello"))
2. The string gets copied over to the buffer completely (strcmp(buffer, "Hello") == 0)
3. A signal is delivered and the function state freezes, we also stop accepting any new signals until after the handler (we do this for convenience)
4. We execute func("World")
5. Now (strcmp(buffer, "World") == 0) and the buffer is printed out "World".
6. We resume the interrupted function and now print out the buffer once again "World" instead of what the function call originally intended "Hello"

Guaranteeing that your functions are signal handler safe can't be solved by removing shared buffers. You must also think about multithreading and synchronization – what happens when I double lock a mutex? You also have to make sure that each function call is reentrant safe. Suppose your original program was interrupted while executing the library code of malloc. The memory structures used by malloc will be inconsistent. Calling printf, which uses malloc as part of the signal handler, is unsafe and will result in **undefined behavior**. A safe way to avoid this behavior is to set a variable and let the program resume operating. The design pattern also helps us in designing programs that can receive signals twice and operate correctly.

```
int pleaseStop ; // See notes on why "volatile sig_atomic_t" is
    better

void handle_sigint(int signal) {
  pleaseStop = 1;
}

int main() {
  signal(SIGINT, handle_sigint);
  pleaseStop = 0;
  while (!pleaseStop) {
    /* application logic here */
  }
  /* clean up code here */
}
```

The above code might appear to be correct on paper. However, we need to provide a hint to the compiler and the CPU core that will execute the main() loop. We need to prevent compiler optimization. The expression pleaseStop doesn't get changed in the body of the loop, so some compilers will optimize it to true TODO: citation needed. Secondly, we need to ensure that the value of pleaseStop is uncached using a CPU register and instead always read from and written to main memory. The sig_atomic_t type implies that all the bits of the variable can be read or modified as an atomic operation - a single uninterruptible operation. It is impossible to read a value that is composed of some new bit values and old bit values.

By specifying pleaseStop with the correct type volatile sig_atomic_t, we can write portable code where the main loop will be exited after the signal handler returns. The sig_atomic_t type can be as large as an int on most modern platforms but on embedded systems can be as small as a char and only able to represent (-127 to 127) values.

```
volatile sig_atomic_t pleaseStop;
```

Two examples of this pattern can be found in COMP a terminal based 1Hz 4bit computer [3]. Two boolean flags are used. One to mark the delivery of SIGINT (CTRL-C), and gracefully shutdown the program, and the other to mark SIGWINCH signal to detect terminal resize and redraw the entire display.

You can also choose a handle pending signals asynchronously or synchronously. To install a signal handler to asynchronously handle signals, use sigaction. To synchronously catch a pending signal use sigwait which blocks until a signal is delivered or signalfd which also blocks and provides a file descriptor that can be read() to retrieve pending signals.

Sigaction

You should use `sigaction` instead of `signal` because it has better defined semantics. `signal` on different operating system does different things which is **bad**. `sigaction` is more portable and is better defined for threads. You can use system call `sigaction` to set the current handler and disposition for a signal or read the current signal handler for a particular signal.

```
int sigaction(int signum, const struct sigaction *act, struct
    sigaction *oldact);
```

The sigaction struct includes two callback functions (we will only look at the 'handler' version), a signal mask and a flags field -

```
struct sigaction {
  void     (*sa_handler)(int);
  void     (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t  sa_mask;
  int       sa_flags;
};
```

Suppose you stumble upon legacy code that uses `signal`. The following snippet installs `myhandler` as the SIGALRM handler.

```
signal(SIGALRM, myhandler);
```

The equivalent `sigaction` code is:

```
struct sigaction sa;
sa.sa_handler = myhandler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGALRM, &sa, NULL)
```

However, we typically may also set the mask and the flags field. The mask is a temporary signal mask used during the signal handler execution. If the thread serving the signal is interrupted in the middle of a system call, the `SA_RESTART` flag will automatically restart some system calls that otherwise would have returned early with EINTR error. The latter means we can simplify the rest of code somewhat because a restart loop may no longer be required.

```
sigfillset(&sa.sa_mask);
sa.sa_flags = SA_RESTART; /* Restart functions if interrupted by
    handler */
```

It is often better to have your code check for the error and restart itself due to the selective nature of the flag.

Blocking Signals

To block signals use `sigprocmask`! With sigprocmask you can set the new mask, add new signals to be blocked to the process mask, and unblock currently blocked signals. You can also determine the existing mask (and use it for later) by passing in a non-null value for oldset.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

From the Linux man page of sigprocmask, here are the possible values for `how` TODO: cite.

- `SIG_BLOCK`. The set of blocked signals is the union of the current set and the set argument.
- `SIG_UNBLOCK`. The signals in set are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- `SIG_SETMASK`. The set of blocked signals is set to the argument set.

The sigset type behaves as a set. It is a common error to forget to initialize the signal set before adding to the set.

```
sigset_t set, oldset;
sigaddset(&set, SIGINT); // Ooops!
sigprocmask(SIG_SETMASK, &set, &oldset)
```

Correct code initializes the set to be all on or all off. For example,

```
sigfillset(&set); // all signals
sigprocmask(SIG_SETMASK, &set, NULL); // Block all the signals
    which can be blocked

sigemptyset(&set); // no signals
sigprocmask(SIG_SETMASK, &set, NULL); // set the mask to be empty
    again
```

If you block a signal with either sigprocmask or pthread_sigmask, then the handler registered with sigaction is not delivered unless explicitly sigwait’ed on TODO: cite.

Sigwait

Sigwait can be used to read one pending signal at a time. sigwait is used to synchronously wait for signals, rather than handle them in a callback. A typical use of sigwait in a multi-threaded program is shown below. Notice that the thread signal mask is set first (and will be inherited by new threads). The mask prevents signals from being *delivered* so they will remain in a pending state until sigwait is called. Also notice the same set sigset_t variable is used by sigwait - except rather than setting the set of blocked signals it is used as the set of signals that sigwait can catch and return.

One advantage of writing a custom signal handling thread (such as the example below) rather than a callback function is that you can now use many more C library and system functions safely.

Based on sigmask code [2]

```
static sigset_t signal_mask; /* signals to block */

int main(int argc, char *argv[]) {
  pthread_t sig_thr_id; /* signal handler thread ID */
  sigemptyset (&signal_mask);
  sigaddset (&signal_mask, SIGINT);
  sigaddset (&signal_mask, SIGTERM);
  pthread_sigmask (SIG_BLOCK, &signal_mask, NULL);

  /* New threads will inherit this thread’s mask */
  pthread_create (&sig_thr_id, NULL, signal_thread, NULL);

  /* APPLICATION CODE */
  ...
}

void *signal_thread(void *arg) {
  int sig_caught;

  /* Use the same mask as the set of signals that we’d like to know
      about! */
  sigwait(&signal_mask, &sig_caught);
  switch (sig_caught) {
    case SIGINT:
    ...
    break;
    case SIGTERM:
    ...
    break;
    default:
    fprintf (stderr, "\nUnexpected signal %d\n", sig_caught);
    break;
  }
}
```

Signals in Child Processes and Threads

This is a recap of the processes chapter. After forking, the child process inherits a copy of the parent's signal dispositions and a copy of the parent's signal mask. If you have installed a SIGINT handler before forking, then the child process will also call the handler if a SIGINT is delivered to the child. If SIGINT is blocked in the parent, it will be blocked in the child as well. Note that pending signals for the child are *not* inherited during forking. After exec though, only the signal mask and pending signals are carried over [1]. Signal handlers are reset to their original action, because the original handler code may have disappeared along with the old process.

Each thread has its own mask. A new thread inherits a copy of the calling thread's mask. On initialization, the calling thread's mask is the exact same as the processes mask. After a new thread is created though, the processes signal mask turns into a gray area. Instead, the kernel likes to treat the process as a collection of threads, each of which can institute a signal mask and receive signals. To start setting your mask, you can use,

```
pthread_sigmask(...); // set my mask to block delivery of some
    signals
pthread_create(...); // new thread will start with a copy of the
    same mask
```

Blocking signals is similar in multi-threaded programs to single-threaded programs with the following translation.

1. Use pthread_sigmask instead of sigprocmask

2. Block a signal in all threads to prevent its asynchronous delivery

The easiest method to ensure a signal is blocked in all threads is to set the signal mask in the main thread before new threads are created.

```
sigemptyset(&set);
sigaddset(&set, SIGQUIT);
sigaddset(&set, SIGINT);
pthread_sigmask(SIG_BLOCK, &set, NULL);

// this thread and the new thread will block SIGQUIT and SIGINT
pthread_create(&thread_id, NULL, myfunc, funcparam);
```

Just as we saw with `sigprocmask`, `pthread_sigmask` includes a 'how' parameter that defines how the signal set is to be used:

```
pthread_sigmask(SIG_SETMASK, &set, NULL) - replace the thread's
    mask with given signal set
pthread_sigmask(SIG_BLOCK, &set, NULL) - add the signal set to the
    thread's mask
pthread_sigmask(SIG_UNBLOCK, &set, NULL) - remove the signal set
    from the thread's mask
```

A signal then can be delivered to any signal thread that is willing to accept that signal. If the two or more threads can receive the signal then which thread will be interrupted is arbitrary! A common practice is to have one thread that can receive all signals or if there is a certain signal that requires special logic, have multiple threads for multiple signals. Even though programs from the outside can't send signals to specific threads, you can do that internally with `pthread_kill(pthread_t thread, int sig)`. In the example below, the newly created thread executing `func` will be interrupted by `SIGINT`

```
pthread_create(&tid, NULL, func, args);
pthread_kill(tid, SIGINT);
pthread_kill(pthread_self(), SIGKILL); // send SIGKILL to myself
```

As a word of warning `pthread_kill(threadid, SIGKILL)` will kill the entire process. Though individual threads can set a signal mask, the signal disposition is *per-process* not *per-thread*. This means `sigaction` can be called from any thread because you will be setting a signal handler for all threads in the process.

The Linux man pages discuss signal system calls in section 2. There is also a longer article in section 7 (though not in OSX/BSD):

```
man -s7 signal
```

Topics

- Signals
- Signal Handler Safety
- Signal Disposition
- Signal States
- Pending Signals when Forking/Exec

- Signal Disposition when Forking/Exec
- Raising Signals in C
- Raising Signals in a multithreaded program

Questions

- What is a signal?
- How are signals served under UNIX? (Bonus: How about Windows?)
- What does it mean that a function is signal handler safe? How about reentrant?
- What is a process signal disposition? How does it differ from a mask?
- What function changes the signal disposition in a single threaded program? How about a multithreaded program?
- What are some drawbacks to using signals?
- What are the ways of asynchronously and synchronously catching a signal?
- What happens to pending signals after a fork? exec? How about my signal mask? How about signal disposition?
- What is the process the kernel goes through from creation to delivery/block?

Bibliography

[1] Executing a file. URL https://www.gnu.org/software/libc/manual/html_node/Executing-a-File.html#Executing-a-File.

[2] pthread$_sigmask.URL.$

Jure orn. gto76/comp-cpp, Jun 2015. URL https://github.com/gto76/comp-cpp/blob/1bf9a77eaf8f57f7358a316e5
src/output.c.

14 Security

Hackers Are Like Artists, Who Wake Up In A Good Mood & Start Painting

Vladimir Putin

Computer security is the protection of hardware and software from unauthorized access or modification. Even if you don't work directly in the computer security field, the concepts are important to learn because all systems will have attackers given enough time. Even though this is introduced as a different chapter, it is important to note that most of these concepts and code examples have already been introduced at different points in the course. We won't go in depth about all of the common ways of attack and defense nor will we go into how to perform all of these attacks in an arbitrary system. Our goal is to introduce you to the field of making programs do what you want to do.

Security Terminology and Ethics

There is some terminology that needs to be explained to get someone who has little to no experience in computer security up to speed

1. An **Attacker** is typically the user who is trying to break into the system. Breaking into the system means performing an action that the developer of the system didn't intend. It could also mean accessing a system you shouldn't have access to.
2. A **Defender** is typically the user who is preventing the attacker from breaking into the system. This may be the developer of the system.
3. There are different types of attackers. There are **white hat** hackers who attempt to hack a defender with their consent. This is commonly a form of pre-emptive testing – in case a not-so-friendly attack comes along. The **black hat** hackers are hackers who hack without permission and the intent to use the information obtained for any purpose. Gray hat hacking differs because the hacker's intent is to inform the defender of the vulnerability – though this can be hard to judge at times.

Danger Will Robinson Before we let you go much further, it is important that we talk about ethics. Before you skip over this section, know that your career quite literally can be terminated over an unethical decision that you might make. The computer fraud and security act is a broad, and arguably terrible law, that casts any non-authorized use of a ‘protected computer’ of a computer as a felony. Since most computers are involved in some interstate/international commerce (the internet) most computers fall under this category. It is important to think about your actions and have some ladder of accountability before executing any attack or defense. To be more concrete, make sure supervisors in your organization have given you their blessing before trying to execute an attack.

First if at all possible, get written permission from one of your superiors. We do realize that this is a cop-out and this puts the blame up a level, but at the risk of sounding cynical organizations will often put blame on an individual employee to avoid damages TODO: Citation Needed. If not possible, try to go through the engineering steps

1. Figure out what the problem is that you are trying to solve. You can’t solve a problem that you don’t fully understand.
2. Determine whetheryou need to “hack” the system. A hack is defined generally as trying to use a system unintendedly. First, you should determine if your use is intended or unintended or somewhere in the middle – get a decision for them. If you can’t get that, make a reasonable judgement as to what the intended use.
3. Figure out a reasonable estimate of what the cost is to “hacking” the system. Get that reasonable estimate checked out with a few engineers so they can highlight things that you may have missed. Try to get someone to sign off on the plan.
4. Execute the plan with caution. If at any point something seems wrong, weigh the risks and execute the plan.

If there isn’t a certain ethical guideline for the current application, then create some. This is often called a `policy vacuum`. This may seem like busy work and more on the “business side” than computer scientists are used to, but your career is at stake here. It is up to you as a computing professional to assess the risk and to decide whether to execute. Courts generally like sitting on precedent, but you can easily say that you aren’t a legal scholar. In lieu, you must be able to say that you reacted as a “reasonable” engineer would react.

TODO: Link to some case studies of real engineers having to decide

CIA Triad

There are three commonly accepted goals to help understand if a system is secure.

1. Information Confidentiality means that only authorized parties are allowed to see a piece of information
2. Information Integrity means that only authorized parties are allowed the modify a piece of information, regardless of whether they are allowed to see it. It ensures that information remains in complete during transit.
3. Information Availability means information, or a service, is available when it is needed.
4. The triad above forms the Confidentiality, Integrity, and Availability (CIA) triad, often authenticity is added as well.

If any of these are broken, the security of a system (either a service or piece of information) has been compromised.

Security in C Programs

Stack Smashing

Consider the following code snippet

```
void greeting(const char *name) {
  char buf[32];
  strcpy(buf, name);
  printf("Hello, %s!\n", buf);
}

int main(int argc, char *argv[]) {
  if (argc < 2){
    return 1;
  }
  greeting(argv[1]);
  return 0;
}
```

There is no checking on the bounds of strcpy! This means that we could potentially pass in a large string and get the program to do something unintended, usually via replacing the return address of the function with the address of malicious code. Most strings will cause the program to exit with a segmentation fault

```
$ ./a.out john
Hello, john!
$ ./a.out JohnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
Program received signal SIGSEGV, Segmentation fault.
...
```

If we manipulate the bytes in certain ways and the program was compiled with the correct flags, we can actually get access to a shell! Consider if that file is owned by root, we put in some valid bytecode (binary instructions) as the string. What will happen is we'll try to execute `execve('/bin/sh', '/bin/sh', NULL , NULL)` that is compiled to the bytecode of the operating system and pass it as part of our string. With some luck, we will get access to a root shell.

```
$ ./a.out <payload>
root#
```

The question arises, which parts of the triad does this break? Try to answer that question yourself. So how would we go about fixing this? We could ingrain into most programmers at the C level to use strncpy or strlcpy on OpenBSD systems. Turning on stack canaries as explained later will fix this issue as well.

Buffer Overflow

Most of you are already familiar with Buffer Overflows! A lot of time they are fairly tame, leading to simple program crashes or funny mistakes. Here is a complete example

```
> cat main.c
#include <stdio.h>

int main() {
    char out[10];
    char in[10];
    fscanf(stdin, "%s", in);
    out[0] = ’a’;
    out[9] = ’\0’;
    printf("%s\n", out);

    return 0;
}
> gcc main.c -fno-stack-protector # need the special flag
    otherwise won’t work
# Stack protectors are explained later.
> ./a.out
hello
a
> ./a.out
helllooooooooo
aoo
>
```

What happens here should be clear if you recall the c memory model. Out and in are both next to each other in memory. If you read in a string from standard input that overflows in, then you end up printing aoo. It gets a little more serious if the snippet starts out as

```
int main() {
  char pass_hash[10];
  char in[10];
  read_user_password(pass_hash, 10);
  // ...
}
```

Out of order instructions & Spectre

Out of order execution is an amazing development that has been recently adopted by many hardware vendors (think 1990s) TODO: citation needed. Processors now instead of executing a sequence of instructions (let's say assigning a variable and then another variable) execute instructions before the current one is done [1, P. 45]. This is because modern processors spend a lot of time waiting for memory accesses and other I/O driven applications. This means that a processor, while it is waiting for an operation to complete, will execute the next few operations. If any of the operations would possibly alter the final result, there is a barrier, or if the re-ordering violates the data dependencies of the instructions, the processor keep the instructions in the stated order [1, P. 296].

Naturally, this allowed CPUs to become more energy-efficient while executing more instructions in real-time and increased security risks from complex architectures. What system programmers are worried about is that operation with mutex locks among threads are out of order – meaning that a pure software implementation of a mutex will fail without copious memory barriers. Therefore, the programmer has to acknowledge that updates may be missed among a series of threads, given that there is no barrier, on modern processors.

One of the most prominent bugs concerning this is Spectre [2]. Spectre is a bug where instructions that otherwise wouldn't be executed are speculatively executed due to out-of-order instruction execution. The following snippet is a high-level proof of concept.

```
char *a[10];
for (int i = 10; i != 1; --i) {
  a[i] = calloc(1, 1);
}
a[0] = 0xCAFE;
int val;
int j = 10; // This will be in a register
int i = 10; // This will be in main memory
for (int i = 10; i != 0; --i, --j) {
  if (i) {
    val = *a[j];
  }
}
```

Let's analyze this code. The first loop allocates 9 elements through a valid malloc. The last element is `0xCAFE`, meaning a dereference should result in a SEGFAULT. For the first 9 iterations, the branch is taken and `val` is assigned to a valid value. The interesting part happens in the last iteration. The resulting behavior of the program is to skip the last iteration. Therefore, `val` never gets assigned to the last value.

But under the right compilation conditions and compiler flags, the instructions will be speculatively executed. The processor thinks that the branch will be taken, since it has been taken in the last 9 iterations. As such, the processor will fetch those instructions. Due to out-of-order instruction execution, while the value of *i* is being fetched from memory, we have to force it not to be in a register. Then, the processor will try to dereference that address. This should result in a SEGFAULT. Since the address was never logically reached by the program, the result is discarded.

Now here is the trick. Even though the value of the calculation would have resulted in a SEGFAULT, the bug doesn't clear the cache that refers to the physical memory where 0xCAFE is located. This is an inexact explanation, but essentially how it works. Since it is still in the cache, if you again trick the processor to read form the cache using `val` then you will read a memory value that you wouldn't be able to read normally. This could include important information such as passwords, payment information, etc.

Operating Systems Security

1. Permissions. In POSIX systems, we have permissions everywhere. There are directories that you can and can't access, files that you can and can't access. Each user account is given access to each file and directory through the read-write-execute (RWX) bits. The user gets matched with either the owner, the group, or 'everyone else', and their access to the file is limited using these bits. Note that permissions work slightly differently on directories compared to files.

2. Capabilities. In addition to permissions on files, each user has a certain set of permissions that they can do. For a full list, you can check capabilities(7). In short, allowing a capability allows a user to perform a set of actions. Some examples include controlling networking devices, creating special files, and peering into IPC or interprocess communication.

3. Address Space Layout Randomization (ASLR). ASLR causes the address spaces of important sections of a process, including the base address of the executable and the positions of the stack, heap and libraries, to start at randomized values, on every run. This is so that an attacker with a running executable has to randomly guess where sensitive information could be hidden. For example, an attacker may use this to easily perform a `return-to-libc` attack.

4. Stack Protectors. Let's say you've programmed a buffer overflow as above. In most cases, what happens? Unless specifically turned off, the compiler will put in stack protectors or stack canaries. This is a value that resides in the stack and must remain constant for the duration of the function call. If that protector is overwritten at the end of the function call, the run time will abort and report to the user that stack smashing was detected.

5. Write xor Execute, also known as `Data Execution Prevention` (DEP). This is a protection that was covered in the IPC section that distinguishes code from data. A page can either be written to or executed but not both. This is to prevent buffer overflows where attackers write arbitrary code, often stored on the stack or heap, and execute with the user's permissions.

6. Firewall. The Linux kernel provides the netfilter module as a way of deciding whetheran incoming connection should be allowed and various other restrictions on connections. This can help with a DDOS attack (explained later).

7. AppArmor. AppArmor is a suite of operating system tools at the userspace level to restrict applications to certain operations.

OpenBSD is an arguably better system for security. It has many security oriented features. Some of these features have been touched upon earlier. An exhaustive list of features is at `https://www.openbsd.org/innovations.html`

1. pledge. Pledge is a powerful command that restricts system calls. This means if you have a simple program like `cat` which only reads to and from files, one can reasonably restrict all network access, pipe access, and write access to files. This is known as the process of "hardening" an executable or system, giving the smallest amount of permissions to the least number of executables needed to run a system. Pledge is also useful in case one tries to perform an injection attack.

2. unveil. Unveil is a system call that restricts the access of a current program to a few directories. Those permissions apply to all forked programs as well. This means if you have a suspicious executable that you want to run whose description is “creates a new file and outputs random words” one could use this call to restrict access to a safe subdirectory and watch it receive the SIGKILL signal if it tries to access system files in the root directory, for example. This could be useful for your program as well. If you want to ensure that no user data is lost during an update (which is what happened with a Steam system update), then the system could only reveal the program’s installation directory. If an attacker manages to find an exploit in the executable, it can only compromise the installation directory.

3. sudo. Sudo is an openBSD project that runs everywhere! Before to run commands as root, one would have to drop to a root shell. Some times that would also mean giving users scary system capabilities. Sudo gives you access to perform commands as root for one-offs without giving a long list of capabilities to all of your users.

Virtualization Security

Virtualization is the act of creating a virtual version of an environment for a program to run on. Though that definition might be bent a little with the advent of new-age bare metal Virtual Machines, the abstraction is still there. One can imagine a single Operating System per motherboard. Virtualization in the software sense is providing “virtual” motherboard features like USB ports or monitors that another program (the bridge) communicates with the actual hardware to perform a task. A simple example is running a virtual machine on your host desktop! One can spin up an entirely different operating system whose instructions are fed through another program and executed on the host system. There are many forms of virtualization that we use today. We will discuss two popular forms below. One form is virtual machines. These programs emulate *all* forms of motherboard peripherals to create a full machine. Another form is containers. Virtual machines are good but are often bulky and programs only need a certain level of protection. Containers are virtual machines that don’t emulate all motherboard peripherals and instead share with the host operating system, adding in additional layers of security.

Now, you can’t have proper virtualization without security. One of the reasons to have virtualization is to ensure that the virtualized environment doesn’t maliciously leak back into the host environment. We say maliciously because there are intended ways of communication that we want to keep in check. Here are some simple examples of security provided through virtualization

1. chroot is a contrived way of creating a virtualization environment. chroot is short for change root. This changes where a program believes that (/) is mounted on the system. For example with chroot, one can make a hello world program believe `/home/bhuvan/` is actually the root directory. This is useful because no other files are exposed. This is contrived because Linux still needs additional tools (think the c standard library) to come from different directories such as `/usr/lib` which means those could still be vulnerable.

2. namespaces are Linux’s better way to create a virtualization environment. We won’t go into this too much, just know that they exist.

3. Hardware virtualization technology. Hardware vendors have become increasingly aware that physical protections are needed when emulating instructions. As such, there can be switches enabled by the user that allows the operating system to flip into a virtualization mode where instructions are run as normal but are monitored for malicious activity. This helps the performance and increases the security of virtualized environments.

Extra: Security through scrubbing

Security isn't just about making sure that a program cannot be manipulated to a malicious user's ends. Some times, security is making sure that the program can't be crashed or effectively timed with input. An example of the former is any attack where the attacker threatens to shut a system down due to a leak. This means that the system has a flaw where the user can input a value and cause the system to fail. One can imagine in mission-critical systems – power grids, medical devices, etc – this isn't a lightly taken threat.

Also, a lot of novice programmers neglect making a program look similar given all inputs. A common example of this is comparing two strings. Let's say that you are guessing the Cross-Site Request Forgery (CSRF) token on a website. If the programmed server returns the request immediately after the token doesn't match the server's, that is a security bug. If the token matches none of the characters, an attacker now know that a quick response means that the first few characters were off. If the token matches some of the characters, an attacker knows that this request will take slightly longer. It is important to balance speed and security. If correct authentication parameters are supplied, the action should succeed immediately. However, if there is a mismatch, there should be no difference in time or content of the response for different reasons of failure, nor should any information be given to the user about the source of error. This is often accomplished by using a standard delay for responses to unsuccessful authentication attempts. We want to use a fast `memcmp`, but it may be insecure.

Cyber Security

Cyber Security is arguably the most popular area of security. More and more of our systems are hacked over the web, it is important to understand how we can protect against these attacks

Security at the TCP Level

1. Encryption. **TCP is unencrypted!** This means any data that is sent over a TCP connection is in plain text. If one needs to send encrypted data, one needs to use a higher level protocol such as HTTPs or develop their own.

2. Identity Verification. In TCP, there is no way to verify the identity of who the program is connecting to. There are no checks or federated databases in place. One just has to trust the DNS server gave a reasonable response which is almost always the incorrect answer. Apart from systems that have an approved white list or a "secret" connection protocol, there is little at the TCP level that one can do to stop.

3. Syn-Ack Sequence Number. This is a security improvement. TCP features what we call sequence numbers. That means that during the SYN-SYN/ACK-ACK dance, a connection starts at a random integer. This is important because if an attacker is trying to spoof packets (pretend those packets are coming from your program) that means that the attacker must either correctly guess – which is hard – or be in the route that your packet takes to the destination – much more likely. ISPs help out with the destination problem because it may send a connection through varying routers which makes it hard for an attacker to sit anywhere and be sure that they will receive your packets – this is why security experts usually advise against using coffee shop wifi for sensitive tasks.

4. Syn-Flood. Before the first synchronization packet is acknowledged, there is no connection. That means a

malicious attacker can write a bad TCP implementation that sends out a flood of SYN packets to a hapless server. The SYN flood is easily mitigated by using IPTABLES or another netfilter module to drop all incoming connections from an IP address after a certain volume of traffic is reached for a certain period.

5. Denial of Service, Distributed Denial of Service is the hardest form of attack to stop. Companies today are still trying to find good ways to ease these attacks. This involves sending all sorts of network traffic forward to servers in the hopes that the traffic will clog them up and slow down the servers. In big systems, this can lead to cascading failures. If a system is engineered poorly, one server's failure causes all the other servers to pick up more work which increases the probability that they will fail and so on and so forth.

Security at the DNS Level

As of 2019, the United States Department of Homeland Security released a directive to switch all services from DNS to DNSSec `https://cyber.dhs.gov/assets/report/ed-19-01.pdf`. This directive is an inherent flaw of the DNS system. First, DNS doesn't offer any sort of verification on domain name requests. That is, it is easy to spoof DNS nameservers such that they point your browser to potentially malicious servers. Remember that DNS requests are sent as unsecured UDP packets, which are prone to tampering. This means that if an attacker snags a plain-text request for a DNS server, that attacker can now send the result back to the requester. More commonly instead of just attacking one person, they will connect to a public wifi station and poison the cache of the router – meaning that all who are connected will get a bad IP address when requesting a domain name. This can get into serious spoofing attacks if one tries to pretend they are a major bank.

Topics

1. Security Terminology
2. Security in local C programs
3. Security in CyberSpace

Review

1. What is a chmod statement to break only the confidentiality of your data?
2. What is a chmod statement to break only the confidentiality and availability of your data?
3. An attacker gains root access on a Linux system that you use to store private information. Does this affect confidentiality, integrity, or availability of your information, or all three?
4. Hackers brute force your git username and password. Who is affected?
5. Why is privilege separation useful in RPC applications?
6. Is it easier to forge a UDP or TCP packet and why?
7. Why are TCP sequence numbers initialized to a random number?

8. What is the impact if the RAM used to hold a shared library (e.g. the C standard library) was writable by any process?

9. Is creating and implementing client-server protocols that are secure and invulnerable to malicious attackers easy?

10. Which is harder to defend against: Syn-Flooding or Distributed Denial of Service?

11. Does deadlock affect the availability of a service?

12. Do buffer overflows / underflows affect the integrity of a data?

13. Why shouldn't stack memory be executable.

14. HeartBleed is an example of what kind of security issue? Which one(s) of the triad does it break?

15. Meltdown and Spectre is an example of what kind of security issue? Which one(s) of the triad does it break?

Bibliography

[1] Part Guide. Intel® 64 and ia-32 architectures software developers manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[2] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

15 Review

A non-comprehensive list of topics is below.

C

Memory and Strings

1. In the example below, which variables are guaranteed to print the value of zero?

```c
int a;
static int b;

void func() {
  static int c;
  int d;
  printf("%d %d %d %d\n",a,b,c,d);
}
```

2. In the example below, which variables are guaranteed to print the value of zero?

```c
void func() {
  int* ptr1 = malloc(sizeof(int));
  int* ptr2 = realloc(NULL, sizeof(int));
  int* ptr3 = calloc(1, sizeof(int));
  int* ptr4 = calloc(sizeof(int), 1);

  printf("%d %d %d %d\n",*ptr1,*ptr2,*ptr3,*ptr4);
}
```

3. Explain the error in the following attempt to copy a string.

```c
char* copy(char*src) {
  char*result = malloc( strlen(src) );
  strcpy(result, src);
  return result;
}
```

4. Why does the following attempt to copy a string sometimes work and sometimes fail?

```c
char* copy(char*src) {
  char*result = malloc( strlen(src) +1 );
  strcat(result, src);
  return result;
}
```

5. Explain the two errors in the following code that attempts to copy a string.

```c
char* copy(char*src) {
  char result[sizeof(src)];
  strcpy(result, src);
  return result;
}
```

6. Which of the following is legal?

```c
char a[] = "Hello"; strcpy(a, "World");
char b[] = "Hello"; strcpy(b, "World12345", b);
char* c = "Hello"; strcpy(c, "World");
```

7. Complete the function pointer typedef to declare a pointer to a function that takes a void* argument and returns a void*. Name your type 'pthread_callback'

```c
typedef ______________________;
```

8. In addition to the function arguments what else is stored on a thread's stack?

9. Implement a version of `char* strcat(char*dest, const char*src)` using only `strcpy` `strlen` and pointer arithmetic

```
char* mystrcat(char*dest, const char*src) {

  ? Use strcpy strlen here

  return dest;
}
```

10. Implement version of size_t strlen(const char*) using a loop and no function calls.

```
size_t mystrlen(const char*s) {

}
```

11. Identify the three bugs in the following implementation of `strcpy`.

```
char* strcpy(const char* dest, const char* src) {
  while(*src) {*dest++ = *src++; }
  return dest;
}
```

Printing

1. Spot the two errors!

```
fprintf("You scored 100%");
```

2. Complete the following code to print to a file. Print the name, a comma and the score to the file 'result.txt'

```
char* name = .....;
int score = ......
FILE *f = fopen("result.txt",_____);
if(f) {
  _____
}
fclose(f);
```

3. How would you print the values of the variables `a`, `mesg`, `val` and `ptr` to a string? Print a as an integer, mesg as C string, val as a double val and ptr as a hexadecimal pointer. You may assume the mesg points to a short C string(<50 characters). Bonus: How would you make this code more robust or able to cope with?

```
char* toString(int a, char*mesg, double val, void* ptr) {
  char* result = malloc( strlen(mesg) + 50);
  -----
  return result;
}
```

Input parsing

1. Why should you check the return value of sscanf and scanf? ## Q 5.2 Why is 'gets' dangerous?
2. Write a complete program that uses `getline`. Ensure your program has no memory leaks.
3. When would you use calloc instead of malloc? When would realloc be useful?
4. What mistake did the programmer make in the following code? Is it possible to fix it

   i) using heap memory? ii) using global (static) memory?

```
static int id;

char* next_ticket() {
  id ++;
  char result[20];
  sprintf(result,"%d",id);
  return result;
}
```

Processes

1. What is a process?
2. What attributes are carried over from a process on fork? How about on a successful exec call?

3. What is a fork bomb? How can we avoid one?
4. What is the wait system call used for?
5. What is a zombie? How do we avoid them?
6. What is an orphan? What happens to them?
7. How do we check the status of a process that has exited?
8. What is a common pattern of processes?

Memory

1. What are the calls in C to allocate memory?
2. What must malloc memory be aligned to? Why is it important?
3. What is Knuth's Allocation Scheme?
4. How would you handle a request in a buddy allocation scheme?
5. What is a free list?
6. What are some different ways of inserting into a free list?
7. What are the benefits and drawbacks to first fit, worst fit, best fit?
8. When would a trivial malloc implementation

    ```
    void *malloc(int size) {
      return (void *)sbrk(size);
    }
    ```

    Be acceptable?

Threading and Synchronization

1. What is a thread? What do threads share?
2. How does one create a thread?
3. Where are the stacks for a thread located in memory?
4. What is a mutex? What problem does it solve?
5. What is a condition variable? What problem does it solve?

6. Write a thread safe linked list that supports insert front, back, pop front, and pop back. Make sure it doesn't busy wait!

7. What is Peterson's Solution to the critical section problem? How about Dekker's?

8. Is the following code thread-safe? Redesign the following code to be thread-safe. Hint: A mutex is unnecessary if the message memory is unique to each call.

```
static char message[20];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *format(int v) {
  pthread_mutex_lock(&mutex);
  sprintf(message, ":%d:" ,v);
  pthread_mutex_unlock(&mutex);
  return message;
}
```

9. Which one of the following may leave a process in running?

   (a) Returning from the pthread's starting function in the last running thread.

   (b) The original thread returning from main.

   (c) Any thread causing a segmentation fault.

   (d) Any thread calling `exit`.

   (e) Calling `pthread_exit` in the main thread with other threads still running.

10. Write a mathematical expression for the number of "W" characters that will be printed by the following program. Assume a,b,c,d are small positive integers. Your answer may use a 'min' function that returns its lowest valued argument.

```
unsigned int a=...,b=...,c=...,d=...;

void* func(void* ptr) {
  char m = * (char*)ptr;
  if(m == 'P') sem_post(s);
  if(m == 'W') sem_wait(s);
  putchar(m);
  return NULL;
}

int main(int argv, char** argc) {
  sem_init(s,0, a);
  while(b--) pthread_create(&tid, NULL, func, "W");
  while(c--) pthread_create(&tid, NULL, func, "P");
  while(d--) pthread_create(&tid, NULL, func, "W");
  pthread_exit(NULL);
  /*Process will finish when all threads have exited */
}
```

11. Complete the following code. The following code is supposed to print alternating `A` and `B`. It represents two threads that take turns to execute. Add condition variable calls to `func` so that the waiting thread need not to continually check the `turn` variable. Q: Is pthread_cond_broadcast necessary or is pthread_cond_signal sufficient?

```
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void* turn;

void* func(void* mesg) {
  while(1) {
    // Add mutex lock and condition variable calls ...

    while(turn == mesg) {
      /* poll again ... Change me - This busy loop burns CPU
          time! */
    }

    /* Do stuff on this thread */
    puts( (char*) mesg);
    turn = mesg;

  }
  return 0;
}

int main(int argc, char** argv){
  pthread_t tid1;
  pthread_create(&tid1, NULL, func, "A");
  func("B"); // no need to create another thread - use the
      main thread
  return 0;
}
```

12. Identify the critical sections in the given code. Add mutex locking to make the code thread safe. Add condition variable calls so that `total` never becomes negative or above 1000. Instead the call should block until it is safe to proceed. Explain why `pthread_cond_broadcast` is necessary.

```c
int total;
void add(int value) {
  if(value < 1) return;
  total += value;
}
void sub(int value) {
  if(value < 1) return;
  total -= value;
}
```

13. An thread unsafe data structure has `size()` `enq` and `deq` methods. Use condition variable and mutex lock to complete the thread-safe, blocking versions.

```c
void enqueue(void* data) {
  // should block if the size() would become greater than 256
  enq(data);
}
void* dequeue() {
  // should block if size() is 0
  return deq();
}
```

14. Your startup offers path planning using the latest traffic information. Your overpaid intern has created a thread unsafe data structure with two functions: `shortest` (which uses but does not modify the graph) and `set_edge` (which modifies the graph).

```c
graph_t* create_graph(char* filename); // called once

// returns a new heap object that is the shortest path from
   vertex i to j
path_t* shortest(graph_t* graph, int i, int j);

// updates edge from vertex i to j
void set_edge(graph_t* graph, int i, int j, double time);
```

For performance, multiple threads must be able to call `shortest` at the same time but the graph can only be modified by one thread when no threads other are executing inside `shortest` or `set_edge`.

15. Use mutex lock and condition variables to implement a reader-writer solution. An incomplete attempt is shown below. Though this attempt is thread safe (thus sufficient for demo day!), it does not allow multiple threads to calculate `shortest` path at the same time and will not have sufficient throughput.

```
path_t* shortest_safe(graph_t* graph, int i, int j) {
  pthread_mutex_lock(&m);
  path_t* path = shortest(graph, i, j);
  pthread_mutex_unlock(&m);
  return path;
}
void set_edge_safe(graph_t* graph, int i, int j, double dist) {
  pthread_mutex_lock(&m);
  set_edge(graph, i, j, dist);
  pthread_mutex_unlock(&m);
}
```

16. How many of the following statements are true for the reader-writer problem?

    - There can be multiple active readers
    - There can be multiple active writers
    - When there is an active writer the number of active readers must be zero
    - If there is an active reader the number of active writers must be zero
    - A writer must wait until the current active readers have finished

Deadlock

1. What do each of the Coffman conditions and what do they mean? Can you provide a definition of each one and an example of breaking them using mutexes?

2. Give a real life example of breaking each Coffman condition in turn. A situation to consider: Painters, paint and paint brushes.

   (a) Hold and wait

   (b) Circular wait

   (c) No preemption

   (d) Mutual exclusion

3. 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 ) ) {/*failed*/
  pthread_mutex_unlock( a );
  ...
}
```

4. How many processes are blocked?

    - P1 acquires R1
    - P2 acquires R2
    - P1 acquires R3
    - P2 waits for R3
    - P3 acquires R5
    - P1 acquires R4
    - P3 waits for R1
    - P4 waits for R5
    - P5 waits for R1

5. What are the pros and cons for the following solutions to dining philosophers

    (a) Arbitrator

    (b) Dijkstra

    (c) Stalling's

    (d) Trylock

IPC

1. What are the following and what is their purpose?

    (a) Translation Lookaside Buffer

    (b) Physical Address

    (c) Memory Management Unit

    (d) The dirty bit

2. How do you determine how many bits are used in the page offset?

3. 20 ms after a context switch the TLB contains all logical addresses used by your numerical code which performs main memory access 100% of the time. What is the overhead (slowdown) of a two-level page table compared to a single-level page table?

4. Explain why the TLB must be flushed when a context switch occurs (i.e. the CPU is assigned to work on a different process).

5. Fill in the blanks to make the following program print 123456789. If `cat` is given no arguments it simply prints its input until EOF. Bonus: Explain why the `close` call below is necessary.

```
int main() {
  int i = 0;
  while(++i < 10) {
    pid_t pid = fork();
    if(pid == 0) {/* child */
      char buffer[16];
      sprintf(buffer, ______,i);
      int fds[ ______];
      pipe(fds);
      write(fds[1], ______,______ ); // Write the buffer into
          the pipe
      close(______);
      dup2(fds[0], ______);
      execlp("cat", "cat", ______);
      perror("exec"); exit(1);
    }
    waitpid(pid, NULL, 0);
  }
  return 0;
}
```

6. Use POSIX calls `fork` `pipe` `dup2` and `close` to implement an autograding program. Capture the standard output of a child process into a pipe. The child process should `exec` the program `./test` with no additional arguments (other than the process name). In the parent process read from the pipe: Exit the parent process as soon as the captured output contains the ! character. Before exiting the parent process send SIGKILL to the child process. Exit 0 if the output contained a !. Otherwise if the child process exits causing the pipe write end to be closed, then exit with a value of 1. Be sure to close the unused ends of the pipe in the parent and child process

7. This advanced challenge uses pipes to get an "AI player" to play itself until the game is complete. The program `tic tac toe` accepts a line of input - the sequence of turns made so far, prints the same sequence followed by another turn, and then exits. A turn is specified using two characters. For example "A1" and "C3" are two opposite corner positions. The string `B2A1A3` is a game of 3 turns/plys. A valid response is `B2A1A3C1` (the C1 response blocks the diagonal B2 A3 threat). The output line may also include a suffix `-I win` `-You win` `-invalid` or `-draw` Use pipes to control the input and output of each child process created. When the output contains a `-`, print the final output line (the entire game sequence and the result) and exit.

8. Write a function that uses fseek and ftell to replace the middle character of a file with an 'X'

```
void xout(char* filename) {
  FILE *f = fopen(filename, ____ );

  // Your code here ...
}
```

9. What is an MMU? What are the drawbacks to using it versus a direct memory system?

10. What is a pipe?

11. What are the pros and cons between named and unnamed pipes?

Filesystems

1. What is the file API?
2. Where are the names of the files stored?
3. What is contained in an inode?
4. What are the two special file names in every directory
5. How do you resolve the following path `a/../b/./c/../../c`
6. What are the rwx groups?
7. What is an UID? GID? What is the difference between UID and Effective UID?
8. What is umask?
9. What is the sticky bit?
10. What is a virtual file system?
11. What is RAID?
12. In an `ext2` filesystem how many inodes are read from disk to access the first byte of the file `/dir1/subdirA/notes.txt` ? Assume the directory names and inode numbers in the root directory (but not the inodes themselves) are already in memory.
13. In an `ext2` filesystem what is the minimum number of disk blocks that must be read from disk to access the first byte of the file `/dir1/subdirA/notes.txt` ? Assume the directory names and inode numbers in the root directory and all inodes are already in memory.
14. In an `ext2` filesystem with 32 bit addresses and 4KiB disk blocks, an inode can store 10 direct disk block numbers. What is the minimum file size required to require a single indirection table? ii) a double direction table?
15. Fix the shell command `chmod` below to set the permission of a file `secret.txt` so that the owner can read,write,and execute permissions the group can read and everyone else has no access.

```
$ chmod 000 secret.txt
```

Networking

1. What is a socket?

2. What are the different layers of the internet?
3. What is IP? What is an IP address?
4. What is TCP? What is UDP? What are the differences?
5. Create a TCP client that send "Hello" to a server.
6. Create a simple TCP echo server. This is a server that reads bytes from a client until it closes and echoes the bytes back to the client.
7. Create a UDP client that would send a flood of packets to a hostname at argv[1].
8. What is HTTP?
9. What is DNS?
10. Why do we use non-blocking IO for networking?
11. What is an RPC?
12. What is special about listening on port 1000 vs port 2000?
    - Port 2000 is twice as slow as port 1000
    - Port 2000 is twice as fast as port 1000
    - Port 1000 requires root privileges
    - Nothing
13. Describe one significant difference between IPv4 and IPv6?
14. When and why would you use ntohs?
15. If a host address is 32 bits which IP scheme am I most likely using? 128 bits?
16. Which common network protocol is packet based and may not successfully deliver the data?
17. Which common protocol is stream-based and will resend data if packets are lost?
18. What is the SYN ACK ACK-SYN handshake?
19. Which one of the following is NOT a feature of TCP?
    (a) Packet reordering
    (b) Flow control
    (c) Packet retranmission
    (d) Simple error detection
    (e) Encryption
20. What protocol uses sequence numbers? What is their initial value? And why?
21. What are the minimum network calls are required to build a TCP server? What is their correct order?
22. What are the minimum network calls are required to build a TCP client? What is their correct order?
23. When would you call bind on a TCP client?

24. What is the purpose of socket bind listen accept ?

25. Which of the above calls can block, waiting for a new client to connect?

26. What is DNS? What does it do for you? Which of the CS241 network calls will use it for you?

27. For getaddrinfo, how do you specify a server socket?

28. Why may getaddrinfo generate network packets?

29. Which network call specifies the size of the allowed backlog?

30. Which network call returns a new file descriptor?

31. When are passive sockets used?

32. When is epoll a better choice than select? When is select a better choice than epoll?

33. Will `write(fd, data, 5000)` always send 5000 bytes of data? When can it fail?

34. How does Network Address Translation (NAT) work?

35. Assuming a network has a 20ms One Way Transit Time between Client and Server, how much time would it take to establish a TCP Connection?

    (a) 20ms

    (b) 40ms

    (c) 100ms

    (d) 60ms

36. What are some of the differences between HTTP 1.0 and HTTP 1.1? How many ms will it take to transmit 3 files from server to client if the network has a 20ms transmit time? How does the time taken differ between HTTP 1.0 and HTTP 1.1?

37. Writing to a network socket may not send all of the bytes and may be interrupted due to a signal. Check the return value of `write` to implement `write_all` that will repeatedly call `write` with any remaining data. If `write` returns -1 then immediately return -1 unless the `errno` is `EINTR` - in which case repeat the last `write` attempt. You will need to use pointer arithmetic.

```
// Returns -1 if write fails (unless EINTR in which case it
   recalls write
// Repeated calls write until all of the buffer is written.
ssize_t write_all(int fd, const char *buf, size_t nbyte) {
  ssize_t nb = write(fd, buf, nbyte);
  return nb;
}
```

38. Implement a multithreaded TCP server that listens on port 2000. Each thread should read 128 bytes from the client file descriptor and echo it back to the client, before closing the connection and ending the thread.

39. Implement a UDP server that listens on port 2000. Reserve a buffer of 200 bytes. Listen for an arriving packet. Valid packets are 200 bytes or less and start with four bytes 0x65 0x66 0x67 0x68. Ignore invalid packets. For valid packets add the value of the fifth byte as an unsigned value to a running total and print the total so far. If the running total is greater than 255 then exit.

Security

1. What are the three measures for data security?
2. What is stack smashing?
3. What is buffer overflows?
4. How does an operating system provide security? What are some examples from Networking and Filesystems?
5. What security features does TCP provide?
6. Is DNS secure?

Signals

1. Give the names of two signals that are normally generated by the kernel
2. Give the name of a signal that can not be caught by a signal
3. Why is it unsafe to call any function (something that it is not signal handler safe) in a signal handler?
4. Write brief code that uses SIGACTION and a SIGNALSET to create a SIGALRM handler.
5. What is the difference between a disposition, mask, and pending signal set?
6. What attributes are passed over to process children? How about exececuted processes?

16 Honors topics

If I have seen further it is by standing on the sholders [sic] of Giants

Sir Isaac Newton

This chapter contains the contents of some of the honors lectures (CS 296-41). These topics are aimed at students who want to dive deeper into the topics of CS 241.

The Linux Kernel

Throughout the course of CS 241, you become familiar with system calls - the userspace interface to interacting with the kernel. How does this kernel actually work? What is a kernel? In this section, we will explore these questions in more detail and shed some light on various black boxes that you have encountered in this course. We will mostly be focusing on the Linux kernel in this chapter, so please assume that all examples pertain to the Linux kernel unless otherwise specified.

What kinds of kernels are there?

As it stands, most of you are probably familiar with the Linux kernel, at least in terms of interacting with it via system calls. Some of you may also have explored the Windows kernel, which we won't talk about too much in this chapter. or `Darwin`, the UNIX-like kernel for macOS (a derivative of BSD). Those of you who might have done a bit more digging might have also encountered projects such a `GNU HURD` or `zircon`.

Kernels can generally be classified into one of two categories, a monolithic kernel or a micro-kernel. A monolithic kernel is essentially a kernel and all of it's associated services as a single program. A micro-kernel on the other hand is designed to have a *main* component which provides the bare-minimum functionality that a kernel needs. This involves setting up important device drivers, the root filesystem, paging or other functionality that is imperative for other higher-level features to be implemented. The higher-level features (such as a networking stack, other filesystems, and non-critical device drivers) are then implemented as separate programs that can interact with the kernel by some form of IPC, typically RPC. As a result of this design, micro-kernels have traditionally been slower than monolithic kernels due to the IPC overhead.

We will devote our discussion from here onwards to focusing on monolithic kernels and unless specified otherwise, **specifically** the Linux kernel.

System Calls Demystified

System Calls use an instruction that can be run by a program operating in userspace that *traps* to the kernel (by use of a signal) to complete the call. This includes actions such as writing data to disk, interacting directly with hardware in general or operations related to gaining or relinquishing privileges (e.g. becoming the root user and gaining all capabilities).

In order to fulfill a user's request, the kernel will rely on `kernel calls`. Kernel calls are essentially the "public" functions of the kernel - functions implemented by other developers for use in other parts of the kernel. Here is a snippet for a kernel call man page:

```
Name

kmalloc allocate memory
Synopsis
void * kmalloc (  size_t size,
    gfp_t flags);

Arguments

size_t size

    how many bytes of memory are required.
gfp_t flags

    the type of memory to allocate.

Description

kmalloc is the normal method of allocating memory for objects
    smaller than page size in the kernel.

The flags argument may be one of:

GFP_USER - Allocate memory on behalf of user. May sleep.

GFP_KERNEL - Allocate normal kernel ram. May sleep.

GFP_ATOMIC - Allocation will not sleep. May use emergency pools.
    For example, use this inside interrupt handlers.
```

You'll note that some flags are marked as potentially causing sleeps. This tells us whetherwe can use those flags in special scenarios, like interrupt contexts, where speed is of the essence, and operations that may block or wait for another process may never complete.

Containerization

As we enter an era of unprecedented scale with around 20 billion devices connected to the internet in 2018, we need technologies that help us develop and maintain software capable of scaling upwards. Additionally, as software increases in complexity, and designing secure software becomes harder, we find that we have new constraints imposed on us as we develop applications. As if that wasn't enough, efforts to simplify software distribution and development, like package manager systems can often lead to headaches of their own, leading to broken packages, dependencies that are impossible to resolve and other such environmental nightmares that have become all to common today. While these seem like disjoint problems at first, all of these and more can be solved by throwing `containerization` at the problem.

What is a container?

A container is almost like a virtual machine. In some senses, containers are to virtual machines as threads are to processes. A container is a lightweight environment that shares resources and a kernel with a host machine, while isolating itself from other containers or processes on the host. You may have encountered containers while working with technologies such as `Docker`, perhaps the most well-known implementation of containers out there.

Linux Namespaces

Building a container from scratch

Containers in the wild: Software distribution is a Snap

Bibliography

17 Appendix

Shell

A shell is actually how you are going to be interacting with the system. Before user-friendly operating systems, when a computer started up all you had access to was a shell. This meant that all of your commands and editing had to be done this way. Nowadays, our computers boot up in desktop mode, but one can still access a shell using a terminal.

```
(Stuff) $
```

It is ready for your next command! You can type in a lot of Unix utilities like `ls`, `echo Hello` and the shell will execute them and give you the result. Some of these are what are known as `shell-builtins` meaning that the code is in the shell program itself. Some of these are compiled programs that you run. The shell only looks through a special variable called path which contains a list of colon separated paths to search for an executable with your name, here is an example path.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:
/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

So when the shell executes `ls`, it looks through all of those directories, finds `/bin/ls` and executes that.

```
$ ls
...
$ /bin/ls
```

You can always call through the full path. That is always why in past classes if you want to run something on the terminal you've had to do `./exe` because typically the directory that you are working in is not in the `PATH` variable. The `.` expands to your current directory and your shell executes `<current_dir>/exe` which is a valid command.

Shell tricks and tips

- The up arrow will get you your most recent command
- `ctrl-r` will search commands that you previously ran
- `ctrl-c` will interrupt your shell's process
- `!!` will execute the last command
- `!<num>` goes back that many commands and runs that
- `!<prefix>` runs the last command that has that prefix
- `!$` is the last arg of the previous command
- `!*` is all args of the previous command
- `p̂atŝub` takes the last command and substitutes the pattern pat for the substitution sub
- `cd -` goes to the previous directory
- `pushd <dir>` pushes the current directory on a stack and cds
- `popd` cds to the directory at the top of the stack

What's a terminal?

A terminal is an application that displays the output from the shell. You can have your default terminal, a quake based terminal, terminator, the options are endless!

Common Utilities

1. `cat` concatenate multiple files. It is regularly used to print out the contents of a file to the terminal but the original use was concatenation.

   ```
   $ cat file.txt
   ...
   $ cat shakespeare.txt shakespeare.txt > two_shakes.txt
   ```

2. `diff` tells you the difference between the two files. If nothing is printed, then zero is returned meaning the files are the same byte for byte. Otherwise, the longest common subsequence difference is printed

```
$ cat prog.txt
hello
world
$ cat adele.txt
hello
it's me
$ diff prog.txt prog.txt
$ diff shakespeare.txt shakespeare.txt
2c2
< world
---
> it's me
```

3. `grep` tells you which lines in a file or standard input match a POSIX pattern.

```
$ grep it adele.txt
it's me
```

4. `ls` tells you which files are in the current directory.
5. `cd` this is a shell builtin but it changes to a relative or absolute directory

```
$ cd /usr
$ cd lib/
$ cd -
$ pwd
/usr/
```

6. `man` every system programmers favorite command tells you more about all your favorite functions!
7. `make` executes programs according to a makefile.

Syntactic

Shells have many useful utilities like saving some output to a file using redirection >. This overwrites the file from the beginning. If you only meant to append to the file, you can use ». Unix also allows file descriptor swapping. This means that you can take the output going to one file descriptor and make it seem like it's coming out of another. The most common one is 2>1 which means take the stderr and make it seem like it is coming out of standard out. This is important because when you use > and » they only write the standard output of the file. There are some examples below.

```
$ ./program > output.txt # To overwrite
$ ./program >> output.txt # To append
$ ./program 2>&1 > output_all.txt # stderr & stdout
$ ./program 2>&1 > /dev/null # don't care about any output
```

The pipe operator has a fascinating history. The UNIX philosophy is writing small programs and chaining them together to do new and interesting things. Back in the early days, hard disk space was limited and write times were slow. Brian Kernighan wanted to maintain the philosophy while omitting intermediate files that take up hard drive space. So, the UNIX pipe was born. A pipe takes the stdout of the program on its left and feeds it to the stdin of the program on its write. Consider the command tee. It can be used as a replacement for the redirection operators because tee will both write to a file and output to standard out. It also has the added benefit that it doesn't need to be the last command in the list. Meaning, that you can write an intermediate result and continue your piping.

```
$ ./program | tee output.txt # Overwrite
$ ./program | tee -a output.txt # Append
$ head output.txt | wc | head -n 1 # Multi pipes
$ ((head output.txt) | wc) | head -n 1 # Same as above
$ ./program | tee intermediate.txt | wc
```

The and || operator are operators that execute a command sequentially. only executes a command if the previous command succeeds, and || always executes the next command.

```
$ false && echo "Hello!"
$ true && echo "Hello!"
$ false || echo "Hello!"
```

What are environment variables?

Each process gets its own dictionary of environment variables that are copied over to the child. Meaning, if the parent changes their environment variables it won't be transferred to the child and vice versa. This is important in the fork-exec-wait trilogy if you want to exec a program with different environment variables than your parent (or any other process).

For example, you can write a C program that loops through all of the time zones and executes the date command to print out the date and time in all locals. Environment variables are used for all sorts of programs so modifying them is important.

Stack Smashing

Each thread uses a stack memory. The stack 'grows downwards' - if a function calls another function, then the stack is extended to smaller memory addresses. Stack memory includes non-static automatic (temporary) variables, parameter values, and the return address. If a buffer is too small some data (e.g. input values from the user), then there is a real possibility that other stack variables and even the return address will be overwritten. The precise layout of the stack's contents and order of the automatic variables is architecture and compiler dependent. With a little investigative work, we can learn how to deliberately smash the stack for a particular architecture.

The example below demonstrates how the return address is stored on the stack. For a particular 32 bit architecture Live Linux Machine, we determine that the return address is stored at an address two pointers (8 bytes) above the address of the automatic variable. The code deliberately changes the stack value so that when the input function returns, rather than continuing on inside the main method, it jumps to the exploit function instead.

```c
// Overwrites the return address on the following machine:
// http://cs-education.github.io/sys/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void breakout() {
  puts("Welcome. Have a shell...");
  system("/bin/sh");
}
void input() {
  void *p;
  printf("Address of stack variable: %p\n", &p);
  printf("Something that looks like a return address on stack:
      %p\n", *((&p)+2));
  // Let's change it to point to the start of our sneaky function.
  *((&p)+2) = breakout;
}
int main() {
  printf("main() code starts at %p\n",main);

  input();
  while (1) {
    puts("Hello");
    sleep(1);
  }

  return 0;
}
```

There are a lot of ways that computers tend to get around this.

Assorted Man Pages

Malloc

```
Copyright (c) 1993 by Thomas Koenig (ig25@rz.uni-karlsruhe.de)
%%%LICENSE_START(VERBATIM)
Permission is granted to make and distribute verbatim copies of this
manual provided the copyright notice and this permission notice are
preserved on all copies.

Permission is granted to copy and distribute modified versions of this
manual under the conditions for verbatim copying, provided that the
entire resulting derived work is distributed under the terms of a.
permission notice identical to this one.

Since the Linux kernel and libraries are constantly changing, this
manual page may be incorrect or out-of-date.  The author(s) assume no
responsibility for errors or omissions, or for damages resulting from
the use of the information contained herein.  The author(s) may not
have taken the same level of care in the production of this manual,
which is licensed free of charge, as they might when working
professionally.

Formatted or processed versions of this manual, if unaccompanied by
the source, must acknowledge the copyright and authors of this work.
%%%LICENSE_END

MALLOC(3)            Linux Programmer's Manual                MALLOC(3)

NAME
       malloc, free, calloc, realloc - allocate and free dynamic memory

SYNOPSIS
       #include <stdlib.h>

       void *malloc(size_t size);
       void free(void *ptr);
       void *calloc(size_t nmemb, size_t size);
       void *realloc(void *ptr, size_t size);
       void *reallocarray(void *ptr, size_t nmemb, size_t size);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       reallocarray():
           _GNU_SOURCE

DESCRIPTION
       The malloc() function allocates size bytes and returns a
       pointer to the allocated memory. The memory is not initialized.
       If size is 0, then malloc() returns either NULL, or
       a unique pointer value that can later be successfully passed
       to free().

       The free() function frees the memory space pointed to by ptr,
       which must have been returned by a previous call to malloc(),
```

calloc(), or realloc(). Otherwise, or if free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.

The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. If nmemb or size is 0, then calloc() returns either NULL, or a unique pointer value that can later be successfully passed to free().

The realloc() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If ptr is NULL, then the call is equivalent to malloc(size), for all values of size; if size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc(), or realloc(). If the area pointed to was moved, a free(ptr) is done.

The reallocarray() function changes the size of the memory block pointed to by ptr to be large enough for an array of nmemb elements, each of which is size bytes. It is equivalent to the call

```
        realloc(ptr, nmemb * size);
```

However, unlike that realloc() call, reallocarray() fails safely in the case where the multiplication would overflow. If such an overflow occurs, reallocarray() returns NULL, sets errno to ENOMEM, and leaves the original block of memory unchanged.

RETURN VALUE

The malloc() and calloc() functions return a pointer to the allocated memory, which is suitably aligned for any built-in type. On error, these functions return NULL. NULL may also be returned by a successful call to malloc() with a size of zero, or by a successful call to calloc() with nmemb or size equal to zero.

The free() function returns no value.

The realloc() function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type and may be different from ptr, or NULL if the request fails. If size was equal to 0, either NULL or a pointer suitable to be passed to free() is returned. If realloc() fails, the original block is left untouched; it is not freed or moved.

On success, the reallocarray() function returns a pointer to the newly allocated memory. On failure, it returns NULL and the original block of memory is left untouched.

ERRORS

calloc(), malloc(), realloc(), and reallocarray() can fail with the following error:

ENOMEM Out of memory. Possibly, the application hit the RLIMIT_AS or RLIMIT_DATA limit described in getrlimit(2).

ATTRIBUTES

For an explanation of the terms used in this section, see attributes(7).

Interface	Attribute	Value
malloc(), free(), calloc(), realloc()	Thread safety	MT-Safe

CONFORMING TO

malloc(), free(), calloc(), realloc(): POSIX.1-2001, POSIX.1-2008, C89, C99.

reallocarray() is a nonstandard extension that first appeared in OpenBSD 5.6 and FreeBSD 11.0.

NOTES

By default, Linux follows an optimistic memory allocation strategy. This means that when malloc() returns non-NULL there is no guarantee that the memory is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer. For more information, see the description of /proc/sys/vm/overcommit_memory and /proc/sys/vm/oom_adj in proc(5), and the Linux kernel source file Documentation/vm/overcommit-accounting.

Normally, malloc() allocates memory from the heap, and adjusts the size of the heap as required, using sbrk(2). When allocating blocks of memory larger than MMAP_THRESHOLD bytes, the glibc malloc() implementation allocates the memory as a private anonymous mapping using mmap(2). MMAP_THRESHOLD is 128 kB by default, but is adjustable using mallopt(3). Prior to Linux 4.7 allocations performed using mmap(2) were unaffected by the RLIMIT_DATA resource limit; since Linux 4.7, this limit is also enforced for allocations performed using mmap(2).

To avoid corruption in multithreaded applications, mutexes are used internally to protect the memory-management data structures employed by these functions. In a multithreaded application in which threads simultaneously allocate and free memory, there could be contention for these mutexes. To scalably handle memory allocation in multithreaded applications, glibc creates additional memory allocation arenas if mutex contention is detected. Each arena is a large region of memory that is internally allocated by the system (using brk(2) or mmap(2)), and managed with its own mutexes.

SUSv2 requires malloc(), calloc(), and realloc() to set errno to ENOMEM upon failure. Glibc assumes that this is done (and the glibc versions of these routines do this); if you use a private

```
       malloc implementation that does not set errno, then certain
       library routines may fail without having a reason in errno.

       Crashes in malloc(), calloc(), realloc(), or free() are almost
       always related to heap corruption, such as overflowing an
       allocated chunk or freeing the same pointer twice.

       The malloc() implementation is tunable via environment
       variables; see mallopt(3) for details.

SEE ALSO
       valgrind(1), brk(2), mmap(2), alloca(3), malloc_get_state(3),
       malloc_info(3), malloc_trim(3), malloc_usable_size(3),
       mallopt(3), mcheck(3), mtrace(3), posix_memalign(3)
```

System Programming Jokes

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

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, it 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…

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.

18 Post Mortems

Hindsight is 20-20

Unknown

This chapter is meant to serve as a big "why are we learning all of this". In all of your previous classes, you were learning what to do. How to program a data structure, how to code a for loop, how to prove something. This is the first class that is largely focused on what *not* to do. As a result, we draw experience from our past in real ways. Sit back and scroll through this chapter as we tell you about the problems of past programmers. Even if you are dealing with something much higher level like web-development, everything relates back to the system.

Shell Shock

Required: Appendix/Shell

This was a back door into most shells. The bug allowed an attacker to exploit an environment variable to execute arbitrary code.

```
$ env x=’() {:;}; echo vulnerable’ bash -c "echo this is a test"
vulnerable...
```

This meant that in any system that uses environment variables and doesn't sanitize their input (hint no one sanitized environment variable input because they saw it as safe) you can execute whatever code you want on other's machines including setting up a web server.

Lessons Learned: On production machines make sure that there is a minimal operating system (something like BusyBox with DietLibc) so that you can understand most of the code in the systems and their effectiveness. Put in multiple layers of abstraction and checks to make sure that data isn't leaked. For example the above is a problem insofar as getting information back to the attackers if it is allowed to communicate with them. This means that you can harden your machine ports by disallowing connections on all but a few ports. Also, you can harden your system to never perform exec calls to perform tasks (i.e. perform an exec call to update a value) and instead do it in C or your favorite programming language of choice. Although you don't have flexibility, you have peace of mind what you allow users to do.

Heartbleed

Required: Intro to C

To put it simply, there were no bounds on buffer checking. The SSL Heartbeat is super simple. A server sends a string of a certain length, and the second server is supposed to send the string of the length back. The problem is someone can maliciously change the size of the request to larger than what they sent (i.e. send "cat" but request 500 bytes) and get crucial information like passwords from the server. There is a Relevant XKCD on it.

Lessons Learned: Check your buffers! Know the difference between a buffer and a string.

Dirty Cow

Required: Processes/Virtual Memory

Dirty Cow

A process usually has access to a set of read-only mappings of memory that if they try to write to they get a segfault. Dirty COW is a vulnerability where a bunch of threads attempts to access the same piece of memory at the same time hoping that one of the threads flips the NX bit and the writable bit. After that, an attacker can modify the page. This can be done to the effective user id bit and the process can pretend it was running as root and spawn a root shell, allowing access to the system from a normal shell.

Lessons Learned: Spinlocks in the kernel are hard.

Meltdown

There is an example of this in the background section

Spectre

Check in the security section.

Mars Pathfinder

Required sections: Synchronization and a bit of Scheduling

Pathfinder Link

The mars pathfinder was a mission that tried to collect climate data on Mars. The finder uses a single bus to communicate with different parts. Since this was 1997, the hardware itself didn't have advanced features like efficient locking so it was up to the operating system developers to regulate that with mutexes. The architecture was pretty simple. There was a thread that controlled data along the information bus, communications thread, and data collection thread in with high, regular, and low priorities with respect to scheduling. The other caveat is that if an interrupt happened at some interval and a task is running and a task is to be scheduled, the task that has the higher priority wins.

The pattern that caused everything to start failing was the data collection thread starts writing to the bus, the information bus thread is waiting on the data. Then out the communication thread comes in to preempt the other lower priority thread **while the lower priority thread still held the mutex**. This means when the regular priority thread tried to lock the bus, the rover would deadlock. After some time the system would reset but isn't good to leave to chance.

Moral of the lesson? Don't have the applications themselves deal with the synchronization. Define a module that handles mutex locking and have the module communicate through files, IPC, etc.

Mars Again

Required Sections: Malloc

Mars

The short of it is that they ran out of memory. The long of it is that they ran out of memory, disk space, and swap space. The moral of the story? Make sure to write code that can handle file failures and can handle files when they close and go out of memory, so the operating system can hot swap files to free up memory. Also clean up files, assume that your temp directory is roughly a hundredth or a thousandth of the total size and use that.

Year 2038

Required sections: Intro to C

2038

This is issue that hasn't happened yet. Unix timestamps are kept as the number of seconds from a particular day (Jan 1st 1970). This is stored as a 32 bit signed integer. In March of 2038, this number will overflow. This isn't a problem for most modern operating system who store 64 bit signed integers which is enough to keep us going until the end of time, but it is a problem for embedded devices that we can't change the internal hardware to. Stay tuned to see what happens.

Lessons learned: Plan like your application will be huge one day.

Northeast Blackout of 2003

Required Sections: Synchronization

2003

A race condition triggered a series of undefined events in a system that caused the blackout of most of the northeastern part of North America for quite some time. This bug also turned off or caused the backup system and logging systems to fail so people didn't even know of the bug for an hour. The exact bits that were flipped are unknown, but patches have been put into place.

Lessons Learned: Modularize your code to localize failures (i.e. keeping race conditions different between processes). If you need to synchronize among processes make sure your failure detection system is not interlaced with your system.

Apple IOS Unicode Handling

Required Sections: Intro to C

Crashing your iphone with text

Wonder why we teach string parsing? Because it is a hard thing to do even for professional software developers. This bug allowed a lot of undefined behavior when trying to parse a series of unicode characters. Apple probably knows why this happened, but our guess is that the parsing of the string happens somewhere inside the kernel and a segfault is reached. When you get a segfault in the kernel, your kernel panics, and the entire device reboots. Undefined behavior means anything though, and a lot of varied things did happen with this bug.

Lessons Learned: Fuzz your kernel

Apple SSL Verification

Required Sections: Intro to C

Apple Bug

Due to a stray goto in Apple's code, a function always returned that an SSL certificate was valid. Naturally, hackers were able to get away with some pretty crazy site names.

Lessons Learned: Always bracket if statements, use gotos sparingly. Chances are if you need to use a goto, write another function or a switch statement with fall throughs (still bad).

Sony Rootkit Installation

Required Sections: Intro to C/Processes

Root Kit Scandal

Picture this. It's 2005, Limewire came a few years prior, the internet was a growing pool of illegal activities – not to say that is fixed now. Sony knew that it didn't have the computing power to police all the interwebs or get around the various technologies people were using to get around copyright protection. So what did they do? With 22 million Music CDs, they required users to install a rootkit on their operating system, so Sony can monitor the device for unethical activities.

Privacy concerns aside, and believe me there are a lot of them, the big problem was that this rootkit is a backdoor for everyone's systems if programmed incorrectly. A rootkit is a piece of code usually installed kernel-side that keeps track of almost anything that a user does. What websites visited, what clicks or keys typed etc. If a hacker finds out about this and there is a way to access that API from the user space level, that means any program can find out important information about your device. Needless to say, people were angry.

Lessons Learned: Get an antivirus and/or apparmor and make sure that an application is only requesting permissions that make sense. If you are torn, try something like Windows sandbox or keep a Sacrificial VM around to see if installing it makes your computer horrible. Don't trust certificates trust code.

Civilization and Ghandi

Required Sections: Intro to C

Ghandi's Aggressiveness

This is probably well known to gamers why someone as (in real life) non-violent as Ghandi was aggressive in the video game civilization. In the original, the game kept aggressiveness as an unsigned integer. During the game, the integer could be decremented and then the problem ensued because Ghandi was already at zero. This caused him to become the most aggressive character in the game.

Lessons Learned: The take away from this is **never** use unsigned numbers unless you have an express written reason for it (reasons include you need to know about the overflow behavior, you are bit shifting, you are bit masking). In every other case, cast it.

The Woes of Shell Scripting

Required Sections: Intro to C/Appending

Steam

There was a simple bug in Steam that caused Steam to remove all of your files in the form of something like this

```
$ ROOT=$(cd $0/; echo $PWD);
$ rm -rf $ROOT
```

What happens if $0 or the first parameter passed into a script doesn't exist? You move to root, and you delete your entire computer.

Lessons Learned: Do parameter checks, always always always `set -e` on a script and if you expect a command to fail, explicitly list it. You can also alias rm to mv and then delete the trash later.

Appnexus Double Free

Required Sections: Intro to C/Malloc

Double Free

Appnexus uses an asynchronous garbage collector that reclaims different parts of the heap when it believes that objects are unused. The architecture is that an element is in the unavailable list and then it is taken out to a to-be-freed list. After a certain time if that element was unused, it is freed and added to the free list. This is fine until two thread try to delete the same object at once, adding to the list twice. After less time, one of the objects was deleted, the delete was announced to other computers.

Lessons Learned: Avoid making hacky software if you need to. Modularize, and set memory limits, and monitor different parts of your code and optimize by hand. There is no general catch-all garbage collector that fits everyone. Even highly tested ones like the JVM need some nudges if you want to get performance out of them.

ATT Cascading Failures - 1990

Required Sections: Intro to C

Explanation

The bug is explained well at the link above. We recommend reading to learn more. A series of network delays that caused some telephone switches across the country to think that other switches were operable when they weren't. When the switches came back online, they realized they had a huge backlog of calls to route and began doing so. Other routing failures and restarts only compounded the problem.

Lessons Learned: Not using C would've actually helped here because of more rigorous fuzzing (though C++ in this day and age would be worse with its language constructs). The real moral of the story is networks are random and expect any jump at any point in your code. That means writing simulations and running them with random delays to figure out bugs before they happen.