8 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 very 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 a 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 not only 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 re-allocate 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 get 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, imagine you had 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 entries. It should be painfully obvious that our lookup scheme would consume all of the memory that we could possibly buy for our 4GB machine. Our lookup table better 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 operating system is 4KiB or 2^{12} addresses, though you 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. A frame actually stores the bytes of interest. To access a particular byte in a frame, you go from the start of the frame and add the offset – discussed later.

A **page table** is a maps 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. You can imagine a page table just as that

```
| page_num | frame_num |
|----------+-----------|
|    0     |     45    |
|    1     |     30    |
```

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.

```
| frame_num |
|-----------|
|     45    | <- 0
|     30    | <- 1
```

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. A process can have. Since there are 2^{20} possible frames, we will need 20 bits to number all of the possible frames meaning `frame_num` 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?

```
                |== Page Number ===|
                |                  |
VirtualAddress = 11110000111100001111000010101010
                                   |          |
                                   |= offset =|
```

We first go to the page table and find the frame number – with pointer arith this would be `page_table_entry *ptr + num`

```
| frame_num |
|-----------|
|    ...    | <- ...
|    101    | <- 11110000111100001111
|    ...    | <- ...
```

Then after, we go to that particular frame and find that byte this would be `char *frame + offset`

```
Frame number 101 in memory

| frame_bytes |
|-------------|
|     ...     | <- 000010101001
|     0x2     | <- 000010101010
|     ...     | <- 000010101011
```

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 look 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 a 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 thats 2^{55} bytes (roughly 40 petabytes). So our page table is too large. In 64 bit architectures 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, not all sub-tables need to exist. An example, two level page table for a 32 bit architecture is shown below.

```
VirtualAddress = 11110000111111110000000010101010 (binary)
                 |-Index1-||        ||          | 10 bit Directory index
                           |-Index2-||          | 10 bit Sub-table index
                                     |--offset--| 12 bit offset (passed directly to RAM)
```

So what is the intuition for dereferencing an address? First, you take the top level page table and find the `Index1`'th entry. That will contain a number that will lead you 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 you add the offset and do the read or write.

Visualizing The Dereference

For those of you who love ascii art or need another way of describing it, let's do it through the scheme as before. First, we find our "base" level pages table. We use that `Index1` to find the first level index.

```
| page_table_num |
|----------------|
|       ...      | <- 111100000
|       402      | <- 111100001 // Index1
|       ...      | <- 111100010
```

Then after we go to the 402'th `page_table`

```
Page table #402

| frame_num |
|-----------|
|    ...    | <- ...
|    703    | <- 1111110000 // Index2
|    ...    | <- ...
```

Same as before, just another level of indirection

```
Frame number #703

| frame_bytes |
|-------------|
|     ...     | <- 000010101001
|     0x5     | <- 000010101010
|     ...     | <- 000010101011
```

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 need 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 address 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 3 page tables of memory or 2KiB for the top level and 4KiB for the two intermediate levels or 10KiB. Here's why. We need at least one frame for the high level directory and two frames for just 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"). Everytime 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 does not have good cache coherence, the TLB will not have the result cache and now the MMU must use the much slower page table to determine the physical frame.

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 permissions 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 an 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 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 you try to write to a read-only page in memory you will get a `SEGFAULT`. That is why sometimes memory accesses segfault and sometimes they don't, it all depends on if your hardware says that you can access.

In addition, 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 runtime 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 will 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 a performance optimization. A page on disk that is paged in to physical memory, then read from, and subsequently paged out again does not need to be written back 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 in to memory. When a dirty bit is not used, 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 is when a running program tries to access some virtual memory in its address space that is not mapped to physical memory. Page faults will also occur in other situations. 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 not in memory but on disk. What this will do is 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 you try 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 you wrote outside the segment that you could write to.

What does this have to do with IPC?

What does this have to do with IPC? Well internally before this point in your programming career you knew that processes had isolation. But, you one didn't really know how that isolation mapped. Two, you may've 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 as mentioned earlier is a trick of virtual memory of instead of mapping a page to just 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 that this does not mean that the file is malloc'ed to memory right away. Take the following code for example.

```
int fd = open(...); //File is 2 Pages
char* addr = mmap(..fd..);
addr[0] = 'l';
```

The kernel may say, see that you want 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 you write to `addr[0]` that you are actually writing to the first byte of the file. The kernel can actually do some optimizations too. Instead of loading the file into memory, it may only load pages at a time because if the file is 1024 pages. You may only access 3 or 4 pages making loading the entire file a waste of time. That is why page faults are so powerful. They let the operating system take control of how much you use your files.

mmap Definitions

mmap does more than just 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 citation 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 you are done `mmap`ping that you `munmap` to tell the operating system that you are no longer using the pages allocated, so the OS can write it back to disk and give you the addresses back 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. It is important because if you believe your operating system may crash or hang with the next few operations, you should create a hard backup of all the operations that have been executed.

The other parameters to mmap are described in the annotated walkthrough below.

Annotated mmap Walkthrough

Below is an annotated walk through of the example code in the man pages

```
#include <sys/mman.h>
#include <sys/stat.h>
```

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); }while (0)

int
main(int argc, char *argv[])
{
    // Various variables that we'll talk about as the code goes through
    char *addr;
    int fd;
    struct stat sb;
    off_t offset, pa_offset;
    size_t length;
    ssize_t s;

    if (argc < 3 || argc > 4) {
        fprintf(stderr, "%s file offset [length]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Open the file and get the size of the file
    fd = open(argv[1], O_RDONLY);
    if (fd == -1)
        handle_error("open");

    if (fstat(fd, &sb) == -1)       /* To obtain file size */
        handle_error("fstat");

    // Grab where the user wants to start the file from
    offset = atoi(argv[2]);

    // Zeros off the bottom N bits of a page (usually 12)
    // mmap works based off of pages so we want to get
    // The first page the user wants
    pa_offset = offset & ~(sysconf(_SC_PAGE_SIZE) - 1);
        /* offset for mmap() must be page aligned */

    if (offset >= sb.st_size) {
        fprintf(stderr, "offset is past end of file\n");
        exit(EXIT_FAILURE);
    }

    if (argc == 4) {
        length = atoi(argv[3]);
        if (offset + length > sb.st_size)
            length = sb.st_size - offset;
                /* Can't display bytes past end of file */

    } else {    /* No length arg ==> display to end of file */
        length = sb.st_size - offset;
    }

    /*
```

```
    Here are the arguments for MMAP
    1. NULL, this tells mmap we don't need any particular address to start
       from
    2. length + offset - pa_offset, mmaps the ''rest'' of the file into
       memory (starting form 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
    */
    addr = mmap(NULL, length + offset - pa_offset, PROT_READ,
                MAP_PRIVATE, fd, pa_offset);
    if (addr == MAP_FAILED)
        handle_error("mmap");

    // Write the file as if a normal buffer
    s = write(STDOUT_FILENO, addr + offset - pa_offset, length);
    if (s != length) {
        if (s == -1)
            handle_error("write");

        fprintf(stderr, "partial write");
        exit(EXIT_FAILURE);
    }

    // This is important! You must mmap the memory out
    munmap(addr, length + offset - pa_offset);

    // This is also important, tells the OS that any fd level changes need
       to be addressed
    close(fd);

    exit(EXIT_SUCCESS);
}
```

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 UNIX, now 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 very slow as it may have to be printed out. The Unix programmers still wanted to create small, portable programs that did one thing very 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 dir_contents
```

What does the following code do? Well it `ls`'s 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 uniq. 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.

```
(0) ls (1)------>(0) cut (1)------->(0) uniq (1)------>(0) sort (1)------>(0) tee (1)
```

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 - you 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 with `write`. A common method of using pipes is to create the pipe before forking in order 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 TA Bhuvan remembered it is you have to *read before you 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 upto 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 */
        }
```

Okay so 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 readers the 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, you could write your own special byte (e.g. 0xff) or a message (`"Bye!"`)

Here's an example of catching this signal that does not work! 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) {
        /* I must be the parent. Close the listening end of the pipe */
        /* I'm not listening anymore!*/
        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 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 and vice versa if you have

two pipes.

The last addendum is that you can actually 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 unrandom. `head` will terminate after 20 newline characters have been read. Well what about `cat`? `cat` needs to receive a SIGPIPE informing it that the program just 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 partial fail if the pipe has a little bit of space left but not enough for the entire message. To avoid this, usually two things are done. 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 write 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 own data intended for the parent (and vice versa)!

Pipes and Dup

A lot of time, 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 take 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 n'th 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.

```c
#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, you read then write (reading is 0 and
        writing is 1)

    if(!fork()) {
        // I am the 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, you need to close all unused ends of pipes otherwise you will deadlock waiting for your processes to finish.

Pipe Conveniences

If you already have a file descriptor then you 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 - just use `fopen` which does the same as `open` and `fdopen` However for pipes, we already have a file descriptor - so this is 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! At the beginning of this course we assumed that file streams are always *line buffered* – the C library will flush its buffer everytime you send a newline character. Actually this is only true for terminal streams - for other filestreams the C library attempts to improve performance by only flushing when it's internal buffer is full or the file is closed.

Even though we have a section on it, it is highly not recommended to fdopen() anything that doesn't point to an underlying file descriptor. The reason being is that while we get conveniences we also get annoyances like the buffering example we mentioned, caching, etc etc. The basic UNIX motto is that anything that you can `fseek` or move to an arbitrary position in, you should be able to `fdopen`. Files satisfiy 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 path name and the operation mode, it will be ready to go! Named pipes take up virtual no space on disk. This means the actual contents of the pipe aren't printed to the file and read from that same file. What the operating system is essentially telling 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 just 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.

Why is my named pipe hanging?

A named pipe `mkfifo` is simply a pipe that you call `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 I do this 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?

```c
//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 you opened the pipe in the first process under both permissions, open won't wait for a reader because you told the operating system that you are a reader! Sometimes it looks like it works because the execution of the code looks something like this.

	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 causes a 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 not available to you, it errors out.
- `read` takes a 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 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 on a valid file descriptor.
- `lseek` takes a file descriptor and moves it to a certain position. Can fail if the seek is out of bounds.
- `fcntl` is the catch all function for file descriptors. You can do everything with this function. Set file locks, read, write, edit permissions, etc ...

The linux interface is very powerful and expressive, but sometimes we need portability for example if we are writing for a mac 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 you can use everywhere, meaning that C on linux uses the above calls.

- `fopen` opens a file and returns an object. `null` is returned if you don'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 file when which you must call `feof()` in order to check.
- `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 you don't get the expressiveness that linux gives you with system calls you can convert back and forth between them with `int fileno(FILE* stream)` and `FILE* fdopen(int fd...)` Another important aspect to note is the C files are **buffered** meaning that their contents may not be written right away by default. You can can change that with C options.

Danger With portability you lose something important, the ability to tell an error. Take for example the advice that we gave you above. You 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 you 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 for you, use the C API of files when you are dealing with a file on disk, which will work fine. Otherwise, be in for a rough ride for portability sake.

How do I tell how large a file is?

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.

But don't do this

Note: This is not recommended in the usual case because of a quirk with the C language. 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 which we know 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 you 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 own file-handle. 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 closed. As such, make sure to close before forking or at least don't trigger a cache inconsistency which is much harder.

So Many IPC Options

Okay so now you have a list of tools in your toolbox in order to tackle communicating between processes, so what should you use?

There really is no hard answer, though this is the most interesting question. Generally we have retained pipes for legacy reasons. This means that we only really 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 as a larger intermediate storage.

mmap is used for two scenarios. One is a `linear` or `near-linear` read through of the file. Meaning, you read the file front to back or back to front. The key is that you don't jump around too much. Jumping around too much causes thrashing and loses all the benefits of using mmap. The other case of mmap is a direct memory inter-process communication. This means that you 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 leveled 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 is the Virtual page number 1, VPN2, VPN3 and the offset?
9. What is a pipe? How do I create a pipe?
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 create 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`.