Inter-Process Communications

Pipes in C

Software Tools & Systems Programming

CSC209H5

Dr. Michael A. Galle

January 24, 2025

Inspired by the lectures of Prof. Robert (Rupert Wu)

Objective

By the end of this class, you will understand:

- The basic concept of pipes for inter-process communications (IPC)
- Their connection to what we already know about file I/O and system calls
- How to create and use pipes in C programs

All materials and code examples are available on the course repo

- git clone https://github.com/michaelgalle/SoftwareSystemsProgramming.git

Overview

1. Bridge-in
2. Intro to Pipes
3. Pipes in C
4. Future directions

1. Bridge-in

- ✓ Shell
- ✓ Processes
- ✓ File system calls
- ✓ Inter-process communications (IPC)
- ❑ Pipes

1. Bridge-in

Recall that a process is

- A particular *instance* of an executing program
- Assigned resources (memory, CPU time) by the system
- Resources (memory) cannot be *shared* between processes

1. Bridge-in

```
int main() {
  pid_t pid = fork();    // Create child process

  if (pid == 0) {        // Child process
    printf("Child process PID: %d\n", getpid());
    execl("/bin/ls", "ls", NULL); // Replace process with ls

  } else if (pid > 0) { // Parent process
      printf("Parent process PID: %d\n", getpid());

  } else {
      perror("fork failed");
  }
  return 0;
}
```

Previous lecture /forkExec

git clone https://github.com/michaelgalle/SoftwareSystemsProgramming.git

Process Creation

- **fork( )** *duplicates* the parent (current) process creating a child, both execute the same code after fork( ).
- **exec*( )** family of functions are used to *replace* the current process with a new one

1. Bridge-in

```
int main() {
    pid_t pid = fork();

    if (pid == 0) {                    // Child process
        printf("Child process running...\n");
        sleep(2);                      // Simulate work
        printf("Child process completed.\n");

    } else if (pid > 0) {              // Parent process
        printf("Parent waiting for child to finish...\n");
        wait(NULL);
        printf("Child completed. Parent resuming.\n");

    } else { perror("fork failed"); }
    return 0;
}
```

Previous lecture /forkWait

Process Synchronization

- **wait( ) / waitpid( )** system calls *wait* for a child or other process to complete

Git clone https://github.com/michaelgalle/SoftwareSystemsProgramming.git

1. Bridge-in

- We have used the following **file** system calls before.

```
int filedesc = open(const char* pathname, int flags, mode_t mode);
```

```
ssize_t write(int filedesc, void* buf, size_t count);
```

```
ssize_t read(int filedesc, const void* buf, size_t count);
```

```
int close(int filedesc);
```

- We have seen how *files* can be used to store and retrieve data

See https://github.com/michaelgalle/SoftwareSystemsProgramming.git /fileIO

1. Bridge-in

- A **file** can *also* be used to send data between *two* processes
- One process can write to the file and the other can read – but this is ***slow***
- Wouldn't it be ***faster*** to skip the data storage and retrieval steps and allow two processes to pass data to each other *directly in memory*?

PIPES

1. Bridge-in

- We have seen UNIX ‘pipes’ (|) in the shell for directing the output of one command to the input of another

```
$ ps aux | sort -nrk 3
```

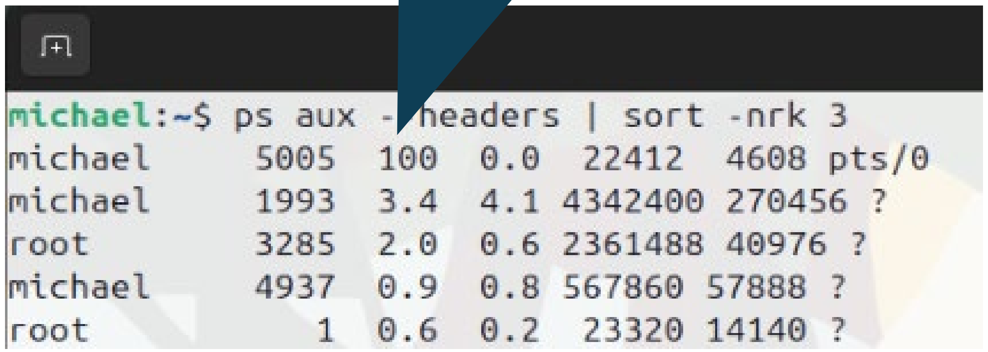
Show %CPU usage of processes

`ps aux`

list processes
‘a’ all
‘u’ by user
‘x’ inc. system

`sort -nrk 3`

‘n’ numerical sort
‘r’ reverse order (highest to lowest)
‘k’ key to sort by
‘k 3’ 3rd column ‘%CPU usage’



- But C pipes are even more powerful …

2. Intro to Pipes

- A pipe enables a **temporary**, **sequential**, **unidirectional** stream of data
- Think of it like a straw with data flows in / out at each end
- A diagram will help us to visualize this …

2. Intro to Pipes

- A pipe has a **read** end at p[0] and a **write** end at p[1].

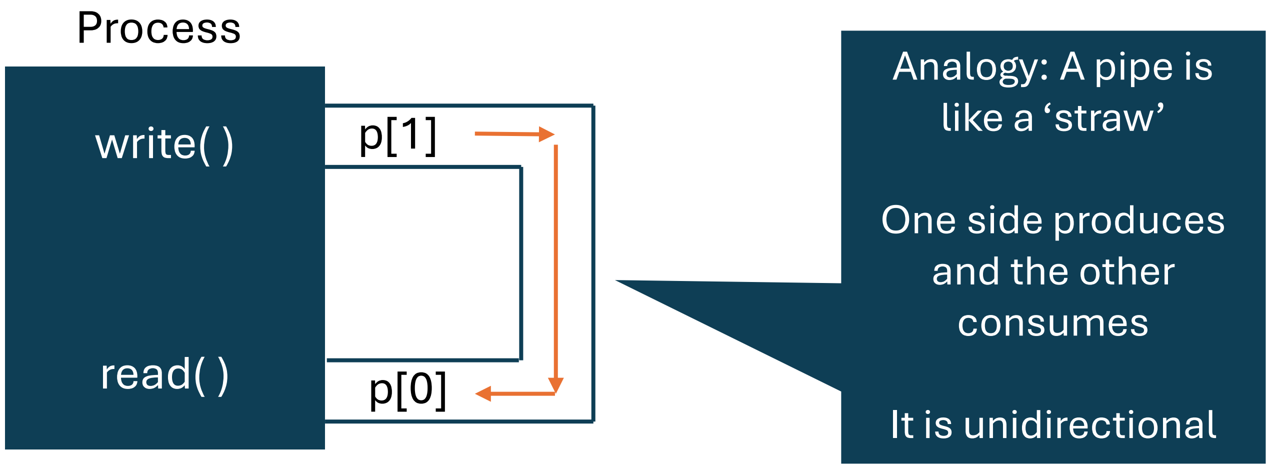


Fig. 1: pipes *to* p[0] and *from* p[1] in a single process

2. Intro to Pipes

If you wanted to send data from a parent to a child process, how would the pipe diagram look?

2. Intro to Pipes

- Pipe from parent to child process

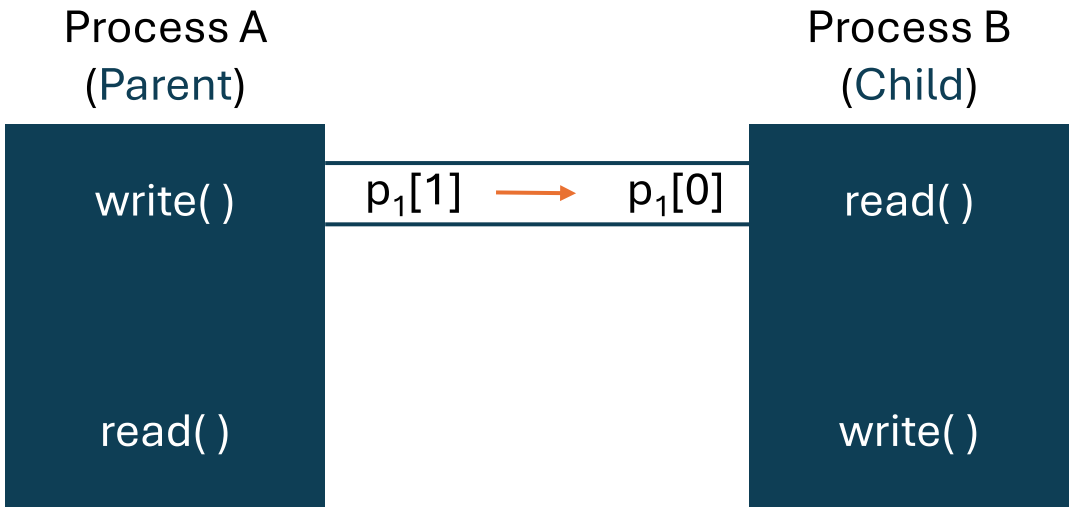


Fig. 2: pipe from parent to child process

2. Intro to Pipes

If you wanted to send data in **both** directions between two processes, how would the pipe diagram look?

2. Intro to Pipes

- Pipes between (sub)processes - both directions

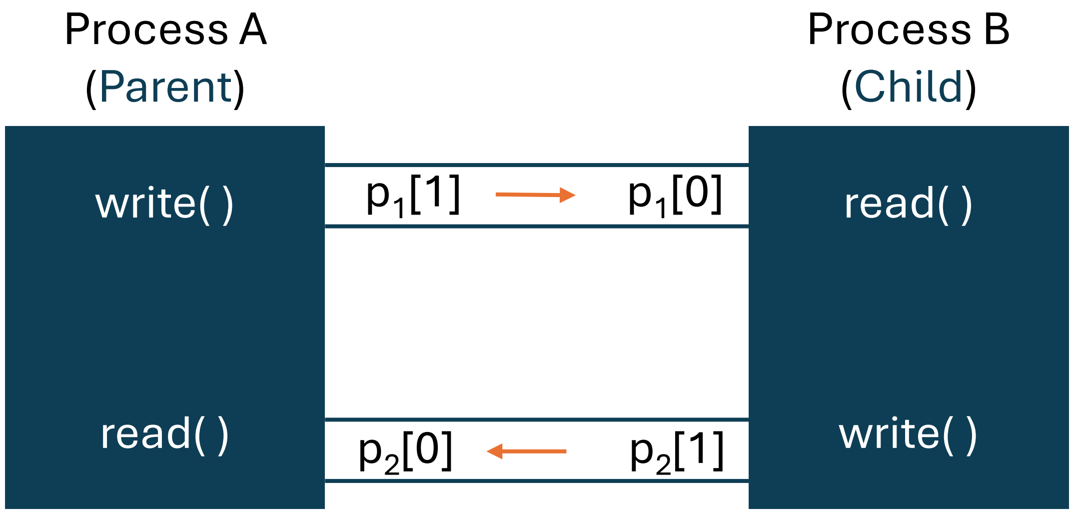


Fig. 3a: Pipes between parent and child process

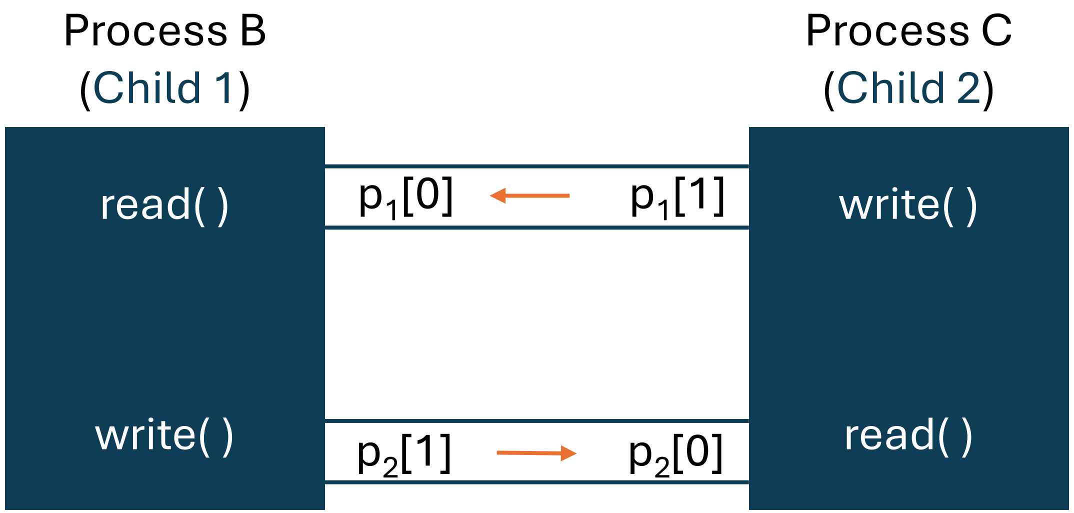


Fig. 3b: Pipes between child processes

3. Pipes in C

- To create a pipe in C, we use the pipe( ) system call

```
int pipe(int filedesc[2]); // fildesc array stores read/write file descriptors
```

- Here is an example of its use

```
#include <unistd.h>
int p[2];      // Array is usually declared on the stack
pipe(p);       // Pass by reference to pipe system call
               // Looks in file descriptor table of the process for two free positions
               // stores filedesc references to them in the p[] array
               // p[0] (stores file descriptor for the read end)
               // p[1] (stores file descriptor for the write end)
```

3. Pipes in C

Danger: Recall 'Defensive programming' lesson, make sure the the call to pipe( ) succeeds

```
#include <unistd.h>
int p[2];
if (pipe(p) < 0) {                  // pipe( ) returns 0 on success
  perror("pipe opening failed");
  exit(1);
}
```

Code on some slides omits this to conserve space.
Code in repo is complete.

3. Pipes in C

- To read and write to a pipe use the system calls seen for file I/O operations

```
ssize_t write(int filedesc, void* buf, size_t count);
```

```
ssize_t read(int filedesc, const void* buf, size_t count);
```

```
int close(int filedesc);
```

- The filedesc used is either p[1] (write) or p[0] (read)
- Let’s see an example of a pipe in a *single* process

3. Pipes in C - Single Process Pipe

```
#define MSG_SIZE 14                  // small static size used
char* msg = “hello, world\n”;        // send buffer for write end of single process pipe

int main() {
  char buf[MSG_SIZE];                // receive buffer for read end of single process pipe
  int p[2];
  if(pipe(p) < 0) { perror(“pipe failed”); exit(1); }
  write(p[1], msg, MSG_SIZE);        // write to pipe at p[1] to a max of MSG_SIZE
  read(p[0], buf, MSG_SIZE);         // read from pipe at p[0] to a max of MSG_SIZE
  write(STDOUT_FILENO, buf, MSG_SIZE);    // prints to console (printf() equivalent)
  close(p[0]); close(p[1]);          // close both pipe ends
  return 0;
}
```

Complete example in folder: /pipes_SPP

Git clone https://github.com/michaelgalle/SoftwareSystemsProgramming.git

3. Pipes in C - Multi-process Pipe

- Communication can be between different processes (e.g. parent / child)

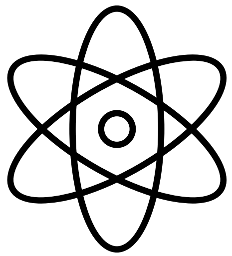
```
#define MSG_SIZE 15
int main() {
    int p[2]; char buf[MSG_SIZE];
    if (pipe(p) < 0) { perror("pipe creation error"); exit(1); }

    pid_t pid = fork();                         // Duplicate process and create child

    if (pid == 0) {                             // Child process: Write to the pipe
        close(p[0]);                            // Close unused read end, child only writes
        write(p[1], "Hello, Parent!", MSG_SIZE); // Child writes
        close(p[1]);                            // Close child write end
    } else if(pid > 0) {                        // Parent process: Read from the pipe
        close(p[1]);                            // Close unused write end in parent, parent only reads
        read(p[0], buf, MSG_SIZE);              // Parent reads from pipe
        write(STDOUT_FILENO, buf, MSG_SIZE);    // Display read buffer contents
        close(p[0]);                            // Close read end of parent
    } else { perror("fork failed"); }
    return 0;
}
```

Complete example in folder /pipes_MPP

Git clone https://github.com/michaelgalle/SoftwareSystemsProgramming.git

3. Pipes in C - Atomicity

- An operation is **atomic** if all the data is sent at once (not split into chunks)



- Read and write operations will be **atomic** if the data size is below a maximum pipe buffer size set by the operating system (e.g., Linux: **64 KB**)

3. Pipes in C - Atomicity

What if the amount of data we want to write exceeds this limit?

3. Pipes in C - Atomicity

- Read and write operations exceeding this limit are not sent atomically and some of the data may be lost
- We can fix this splitting the data into **sequential chunks**
- Let’s explore an example in which a write process (parent) attempts to send **128 kB of data** to a receive process (child) that **can only read in 32 kB chunks**
- We will see how both sides must sequence (track) the bytes in their buffer

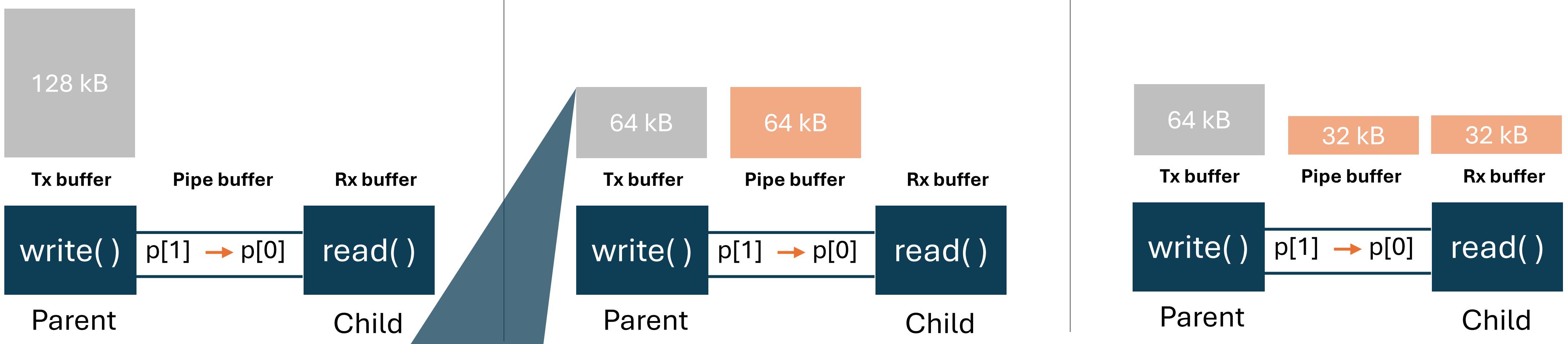
3. Pipes in C - Atomicity

Overview of what will happen in the code ...

1. Parent has 128 kB to Transmit

2. Parent fills pipe buffer to maximum and is stopped to prevent buffer overflow

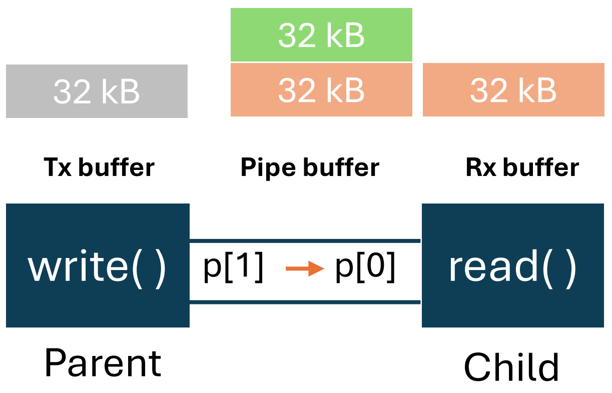
3. Child reads 32 kB chunk, freeing the pipe buffer by 32 kB



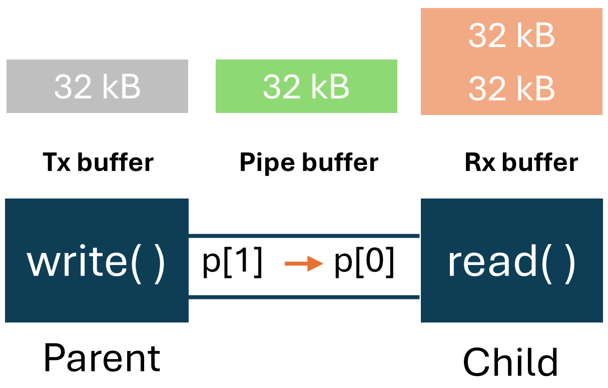
The parent tries to send all 128 kB at once but only the first 64 kB are sent.
The parent must **track** the starting address of the untransmitted bytes in the Tx buffer

3. Pipes in C - Atomicity

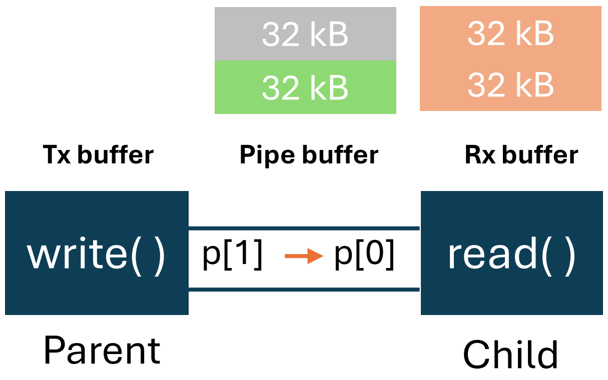
4. Parent sees 32 kB of free space in pipe buffer and transmits next 32 kB



5. Child reads 32 kB chunk, freeing the pipe buffer by 32 kB

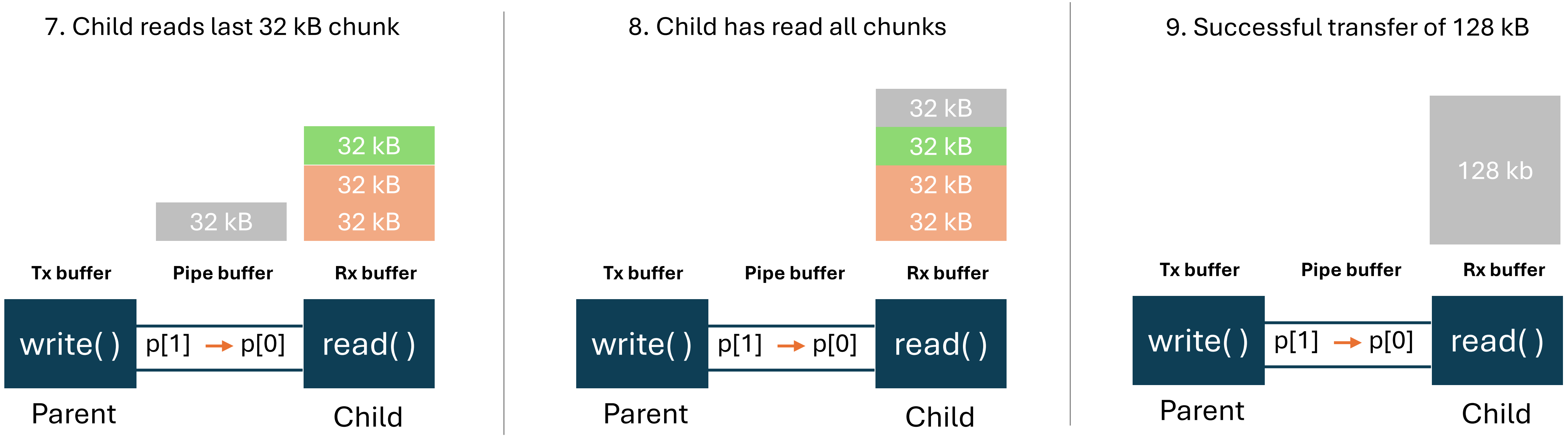


6. Parent sees 32 kB of free space in pipe buffer and transmits next 32 kB



What do you notice about the effective transfer rate?

3. Pipes in C - Atomicity



Effective transfer rate is limited by the read process to 32 kB / chunk

3. Pipes in C - Atomicity

```c
#define TOTAL_SIZE (128 * 1024)  // 128 kB total
#define CHUNK_SIZE  (32 * 1024)  // 32 kB chunks

int main() {
  int p[2]; pipe(p);      // Create pipe
  pid_t pid = fork();     // Child process is dupl. of parent

  if (pid == 0) {
    // Child process (Reader)
    char chunk_buf[CHUNK_SIZE];
    char* received = (char*)malloc(TOTAL_SIZE);  // receive buf
    ssize_t B_read, total = 0;

    while((B_read = read(p[0], chunk_buf, CHUNK_SIZE)) > 0) {
        memcpy(received + total, chunk_buf, B_read);
        total += B_read;
    }
    printf("Child read: %zd bytes in receive buffer\n", total);
    free(received); close(p[0]); close(p[1]);
  }
```

Child only reads from the pipe in 32 kB chunks

```c
 else if(pid > 0) {
   // Parent process (Writer)
   char buf[TOTAL_SIZE];
   memset(buf, 'A', TOTAL_SIZE);
   ssize_t B_written, total = 0;

   while(total < TOTAL_SIZE) {
     B_written = write(p[1], buf + total, TOTAL_SIZE - total);
     if(B_written < 0) { perror("write error"); break; }
     total += B_written;
   }

   printf("Parent wrote a total of %zd bytes\n", total);
   close(p[0]); close(p[1]);
 }
 else { perror("fork failed"); }
return 0;
}
```

First write operation in the loop fills the pipe buffer completely (64 kB) then **stops** to prevent buffer overflow.

The *rest* of the data must be sent in *later* write operations (need to track bytes not yet written)

Since the pipe buffer is freed in 32 kB chunks all *subsequent* data can only be written in 32 kB chunks

Git clone https://github.com/michaelgalle/SoftwareSystemsProgramming.git **/pipes_Atomic**

Summary

- Pipes are useful for efficient inter-process communications
- A pipe enables a temporary, sequential, unidirectional stream of data
- Read / write operations on a pipe use file I/O system calls
- Operations will be atomic if the data size is below a maximum pipe buffer size.
- If the data size is larger, we can always split the data into sequential chunks

4. Future Directions

- Pipes are great for IPC between processes on the **same machine**
- What if we want to enable communication between processes on different machines?

SOCKETS

See https://github.com/michaelgalle/SoftwareSystemsProgramming.git - future lecture - **/sockets**
view: https://www.youtube.com/watch?v=Y3MLI-sHfJk

4. Future Directions

- Processes are useful for isolated tasks (e.g., different applications)
- Inter-process Communication (IPC) is required since memory is **isolated**
- How can we eliminate the need for IPC between two *similar* concurrent tasks?
- By allowing them to **share** memory and other resources

THREADS

See https://github.com/michaelgalle/SoftwareSystemsProgramming.git - future lecture - **/threads**

4. Future Directions

- Pipes are useful for inter-process communications
- But how can two processes (or a process and the operating system) communicate and handle an **asynchronous** event, like an interrupt?

SIGNALS

See https://github.com/michaelgalle/SoftwareSystemsProgramming.git - future lecture - **/signals**

4. Future Directions

Aside …

- Signals are software interrupts delivered to a process
- Examples
  - SIGINT – sent when ‘Ctrl+C’ is pressed
  - SIGPIPE – sent when a process writes to a pipe with no readers (test / error checking)

See https://github.com/michaelgalle/SoftwareSystemsProgramming.git - future lecture - **/signals**