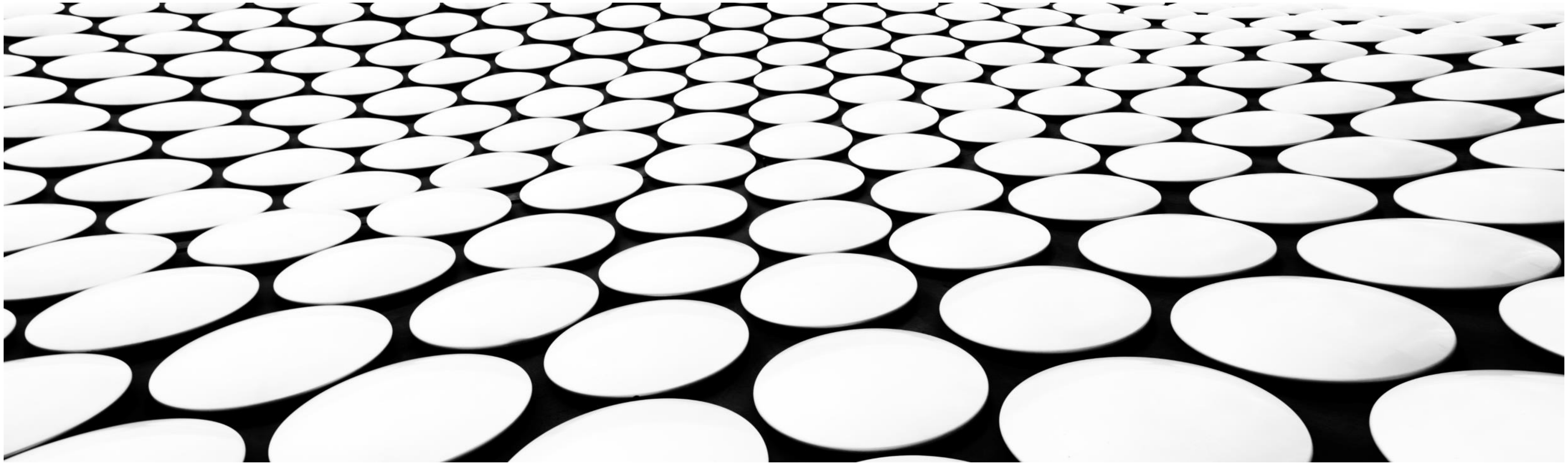
FFMPEG library and systems programming tools

COEN-317: Distributed Systems
Robert Bruce
Department of Computer Science and Engineering
Santa Clara University



Introduction to FFMPEG

FFMPEG is a library and series of command line tools for manipulating video format files. This library is useful for:

- Creating a video movie from a series of still images
- Extracting video metadata information (encoding format, frames per second, number of frames, etc.
- Extracting a series of still images from a video movie.

FFMPEG: determine number of frames

To extract metadata information (number of frames) from an input_video:

```
ffprobe -v error -count_frames -select_streams v:0 -show_entries
stream=nb_read_frames -of default=nokey=1:noprint_wrappers=1
Big_Buck_Bunny_Trailer_1080p.ogx
```

Output:

```
813
```

FFMPEG: determine frame resolution

To extract metadata information (frame resolution) from an input_video:

```
ffprobe -v error -of flat=s=_ -select_streams v:0 -show_entries
stream=height,width Big_Buck_Bunny_Trailer_1080p.ogx
```

Output:

```
streams_stream_0_width=1920

streams_stream_0_height=1080
```

FFMPEG: determine frame rate

To extract metadata information (frames per second) from an input_video:

```
ffprobe -v error -select_streams v:0 -show_entries stream=avg_frame_rate -of default=noprint_wrappers=1:nokey=1 Big_Buck_Bunny_Trailer_1080p.ogx
```

Output:

```
25/1
```

FFMPEG: determine frame rate

To extract metadata information (frames per second) from an input_video:

```
ffprobe -v error -select_streams v:0 -show_entries stream=avg_frame_rate -
of default=noprint_wrappers=1:nokey=1 Big_Buck_Bunny_Trailer_1080p.ogx
```

Output:

```
25/1
```

25 frames

FFMPEG: determine frame rate

To extract metadata information (frames per second) from an input_video:

```
ffprobe -v error -select_streams v:0 -show_entries stream=avg_frame_rate -
of default=noprint_wrappers=1:nokey=1 Big_Buck_Bunny_Trailer_1080p.ogx
```

Output:

```
25/1
```

$\uparrow$

in one second

FFMPEG: convert input video into a series of still images

To extract a series of still images (out.1.png, out.2.png, out.3.png, etc.) from an input video:

```
ffmpeg -i Big_Buck_Bunny_Trailer_1080p.ogx -vf fps=25/1 out.%d.png
```

FFMPEG: create H.264 video from still images in an MPEG4 container

To create an output video (output.mp4) from a series of still images (out.1.png, out.2.png, out.3.png, etc.) using an H.264 codec in an MPEG4 container at 25 frames per second:

```
ffmpeg -r 25/1 -start_number 1 -f image2 -i out.%d.png -c:v libx264
output.mp4
```

FFMPEG: create H.264 video from still images in a Matroska container

To create an output video (output.mkv) from a series of still images (out.1.png, out.2.png, out.3.png, etc.) using an H.264 codec in a Matroska container at 25 frames per second:

```
ffmpeg -r 25/1 -start_number 1 -f image2 -i out.%d.png -c:v libx264
output.mkv
```

FFMPEG: create H.265 video from still images into an MPEG4 container

To create an output video (output.mp4) from a series of still images (out.1.png, out.2.png, out.3.png, etc.) using an H.265 codec in an MPEG4 container at 25 frames per second:

```
ffmpeg -r 25/1 -start_number 1 -f image2 -i out.%d.png -c:v libx265
output.mp4
```

FFMPEG: create H.265 video from still images in a Matroska container

To create an output video (output.mkv) from a series of still images (out.1.png, out.2.png, out.3.png, etc.) using an H.265 codec in a Matroska container at 25 frames per second:

```
ffmpeg -r 25/1 -start_number 1 -f image2 -i out.%d.png -c:v libx265
output.mkv
```

Systems programming functions: fork() and exec()

To invoke the ffmpeg or ffprobe commands above within a C program, you will need to use fork() and exec() functions.

- fork() is used to launch a child process. When a program executes the fork() function, the operating system creates a new child process independent of the parent process that invoked it.
- The exec() command tells the operating system to execute the process specified by exec. The program that launches the exec() command will terminate as the new process specified in the exec() function is executed.

An alternative for fork(), exec(), and pipe: popen()

An alternative to using named pipes, fork(), and exec() is to use the popen() command:

- The popen() function automatically forks a child processes then invokes a command shell to execute the command specified in popen(). The output from the command executed is then passed back through a named pipe to the parent process that launched popen().

Example program: fork_exec_pipe.c

My example C program, *fork_exec_pipe.c* is located on Camino under files folder : example programs : fork, exec, pipe example.

1. This program creates a named pipe then performs a fork() operation.
2. The program is split into two processes: a parent process and a child process.
3. The parent and child processes communicate through the named pipe.
4. The parent process waits for output from the child process.
5. The child process invokes exec() to run an ffprobe command. The output from the ffprobe command is then sent to the parent process.
6. The parent process receives, then stores output from the child process in a buffer.
7. The parent process sends this output to standard output.

Dissecting fork_exec_pipe.c

```
int main (int argc, char *argv[])
{
  char line[2000], master_buffer[10000], input_video_filename[1280];
  int i, fd[2], n, process_status, child_pid, bytes_read, total_bytes_read;
  pid_t wpid;

  total_bytes_read = 0;
  memset (&master_buffer[0], 0, sizeof(master_buffer));
  strcpy (&input_video_filename[0], "Big_Buck_Bunny_Trailer_1080p.ogx");
  if (pipe(fd) < 0)
  {
    printf ("pipe() failed.\n");
    exit (EXIT_FAILURE);
  }
```

Dissecting fork_exec_pipe.c

```
switch (child_pid = fork())
{
  case -1:
    printf ("fork() error\n");
    exit (EXIT_FAILURE);
  break;
  case 0:                // child
    dup2 (fd[1], 1);     // 0 is STDIN (read), 1 is STDOUT (write)!
    dup2 (STDOUT_FILENO, STDERR_FILENO); // redirect standard error to standard out.
    close (fd[0]);       // the child does not need this end of the pipe
    execl ("/usr/bin/ffprobe", "/usr/bin/ffprobe", "-i", &input_video_filename[0], (char *) NULL);
    exit (EXIT_FAILURE); // should not make it here!
  break;
  default:               // parent
    close (fd[1]);       // the parent does not need this end of the pipe
    memset (&line[0], 0, sizeof(line));
    bytes_read = read(fd[0], &line[0], sizeof(line) - 1);
    while (bytes_read > 0)
    {
      line[bytes_read] = '\0';
      total_bytes_read = total_bytes_read + bytes_read;
      if (total_bytes_read < (sizeof(master_buffer) - 1))
      {
        strcat (master_buffer, line);
      }
      else
      {
        printf ("Exceeded master buffer size\n");
      }
      bytes_read = read(fd[0], line, sizeof(line) - 1);
    }
    wpid = waitpid (child_pid, &process_status, 0);
    if (wpid < 0)
    {
      printf ("waitpid error\n");
      exit (EXIT_FAILURE);
    }
  break;
}
```

Dissecting fork_exec_pipe.c

```
  for (i = 0; i < total_bytes_read; i++)
  {
    putchar (master_buffer[i]);
  }
  exit (EXIT_SUCCESS);
}
```

Dissecting fork_exec_pipe.c

```
switch (child_pid = fork())
{
  case -1:
    printf ("fork() error\n");
    exit (EXIT_FAILURE);
  break;
  case 0:                // child
    dup2 (fd[1], 1);     // 0 is STDIN (read), 1 is STDOUT (write)!
    dup2 (STDOUT_FILENO, STDERR_FILENO); // redirect standard error to standard out.
    close (fd[0]);       // the child does not need this end of the pipe
    execl ("/usr/bin/ffprobe", "/usr/bin/ffprobe", "-i", &input_video_filename[0], (char *) NULL);
    exit (EXIT_FAILURE); // should not make it here!
  break;
  default:               // parent
    close (fd[1]);       // the parent does not need this end of the pipe
    memset
    bytes_re
    while (
    {
      line[
      total_bytes_read = total_bytes_read + bytes_read;
      if (total_bytes_read < (sizeof(master_buffer) - 1))
      {
        strcat (master_buffer, line);
      }
      else
      {
        printf ("Exceeded master buffer size\n");
      }
      bytes_read = read(fd[0], line, sizeof(line) - 1);
    }
    wpid = waitpid (child_pid, &process_status, 0);
    if (wpid < 0)
    {
      printf ("waitpid error\n");
      exit (EXIT_FAILURE);
    }
  break;
}
```

fork() has failed. This is a catastrophic failure and indicates the operating system has exhausted all memory to support additional processes. Our only recourse is to exit immediately.

Dissecting fork_exec_pipe.c

```
switch (child_pid = fork())
{
  case -1:
    printf ("fork() error\n");
    exit (EXIT_FAILURE);
  break;
  case 0:                // child
    dup2 (fd[1], 1);     // 0 is STDIN (read), 1 is STDOUT (write)!
    dup2 (STDOUT_FILENO, STDERR_FILENO); // redirect standard error to standard out.
    close (fd[0]);       // the child does not need this end of the pipe
    execl ("/usr/bin/ffprobe", "/usr/bin/ffprobe", "-i", &input_video_filename[0], (char *) NULL);
    exit (EXIT_FAILURE); // should not make it here!
  break;
  default:               // parent
    close (
    memset
    bytes_r
    while (
    {
      line[
      total
      if (t
      {
        str
      }
      else
      {
        pri
      }
      bytes
    }
    wpid = waitpid (child_pid, &process_status, 0);
    if (wpid < 0)
    {
      printf ("waitpid error\n");
      exit (EXIT_FAILURE);
    }
  break;
}
```

This section of the program will be executed in the child process after a *fork()* operation.

Inside the child process, we close the standard input end of the pipe. This is because the child process will only be sending output to the parent process. The child will not be receiving data through the pipe from the parent process. Therefore, close this unused end of the pipe.

Inside the child process, the standard error (which ffmpeg tools such as ffprobe use) is directed to standard output. This way, all standard output and standard error messages will be sent to parent process through the pipe.

Lastly, the child invokes the exec() command to transfer control to ffprobe.

Dissecting fork_exec_pipe.c

The default section of the switch statement section will be executed in the parent process after a *fork()* operation. The parent process will not be sending output through the pipe to the child process. The parent process will only be receiving data from the child process. Therefore, the sending end of the pipe (from parent to child) is closed. The next step is to begin reading data from the pipe (which is being sent from the child process to the parent process). This data is stored in a character buffer in the parent process. The parent process waits for the child process to exit properly then the parent process outputs the character buffer contents to standard output then the parent process exits.

```
switch (child_pid = fork())
{
  case -1:
    printf
    exit (E
  break;
  case 0:
    dup2 (f
    dup2 (S
    close (
    execl (
    exit (E
  break;
  default:               // parent
    close (fd[1]);       // the parent does not need this end of the pipe
    memset (&line[0], 0, sizeof(line));
    bytes_read = read(fd[0], &line[0], sizeof(line) - 1);
    while (bytes_read > 0)
    {
      line[bytes_read] = '\0';
      total_bytes_read = total_bytes_read + bytes_read;
      if (total_bytes_read < (sizeof(master_buffer) - 1))
      {
        strcat (master_buffer, line);
      }
      else
      {
        printf ("Exceeded master buffer size\n");
      }
      bytes_read = read(fd[0], line, sizeof(line) - 1);
    }
    wpid = waitpid (child_pid, &process_status, 0);
    if (wpid < 0)
    {
      printf ("waitpid error\n");
      exit (EXIT_FAILURE);
    }
  break;
}
```

For further reading

FFMPEG source code and documentation
https://www.ffmpeg.org/

My example C program on Camino (under files folder : example programs : fork, exec, pipe example):
fork_exec_pipe.c