

Queueing & File I/O

*CS 241: Systems Programming
Discussion, Week 7*

slides prepared by Sameer Sundresh

LMP1

Queueing

Files Overview

File I/O

LMP1

LMP1 is the first part of a **three-MP set** aimed at implementing a functional **network file system**.

LMP1 deals with **file I/O**, while subsequent MPs add memory management and networking.

In LMP1, you **implement a basic file system**, and **test its performance** using the *Bonnie* filesystem benchmark.

LMP1: four parts

1. Use *Bonnie* with the provided stub code to understand its functionality
2. Implement basic FS operations for our file system (`file_read`, `file_write`, `file_info`).

1 & 2 will be due Wednesday after Spring Break

3. Implement additional FS operations (file creation/deletion, directory ops, checksums)
4. Modify *Bonnie* to use our custom FS implementation and test performance

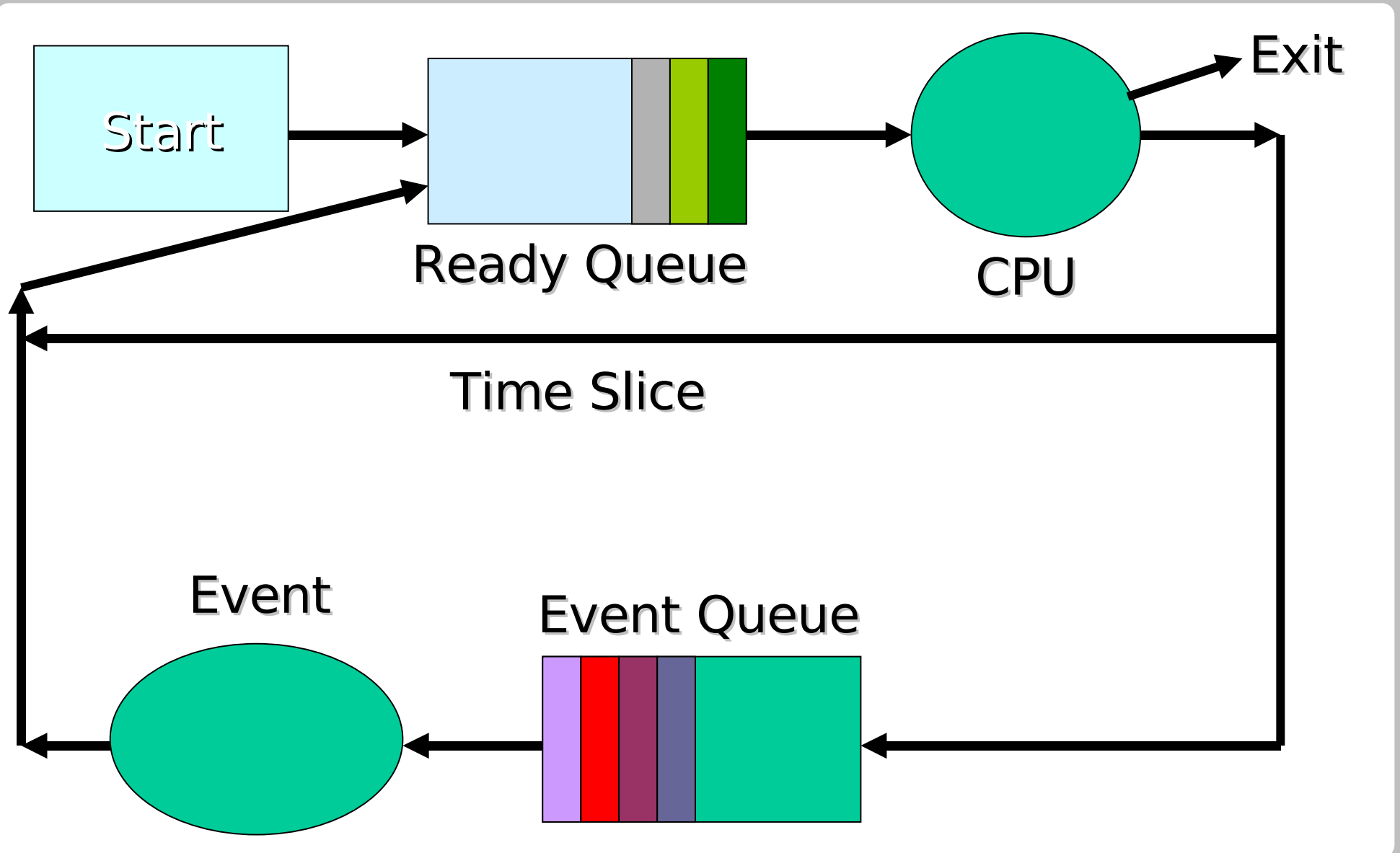
LMP1

Queueing

Files Overview

File I/O

Queuing Diagram for Processes

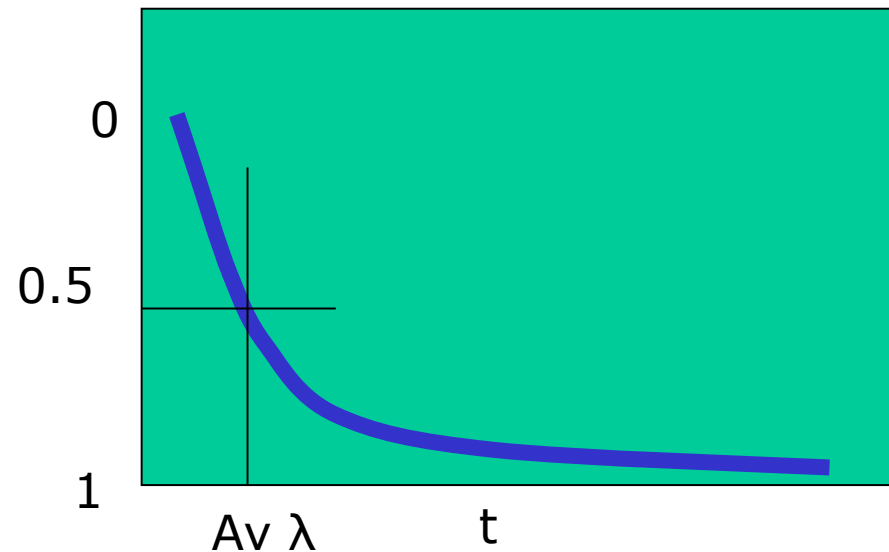


Queueing Theory

Steady state

Poisson arrival with λ constant arrival rate (customers per unit time) each arrival is independent.

$$P(\tau \leq t) = 1 - e^{-\lambda t}$$



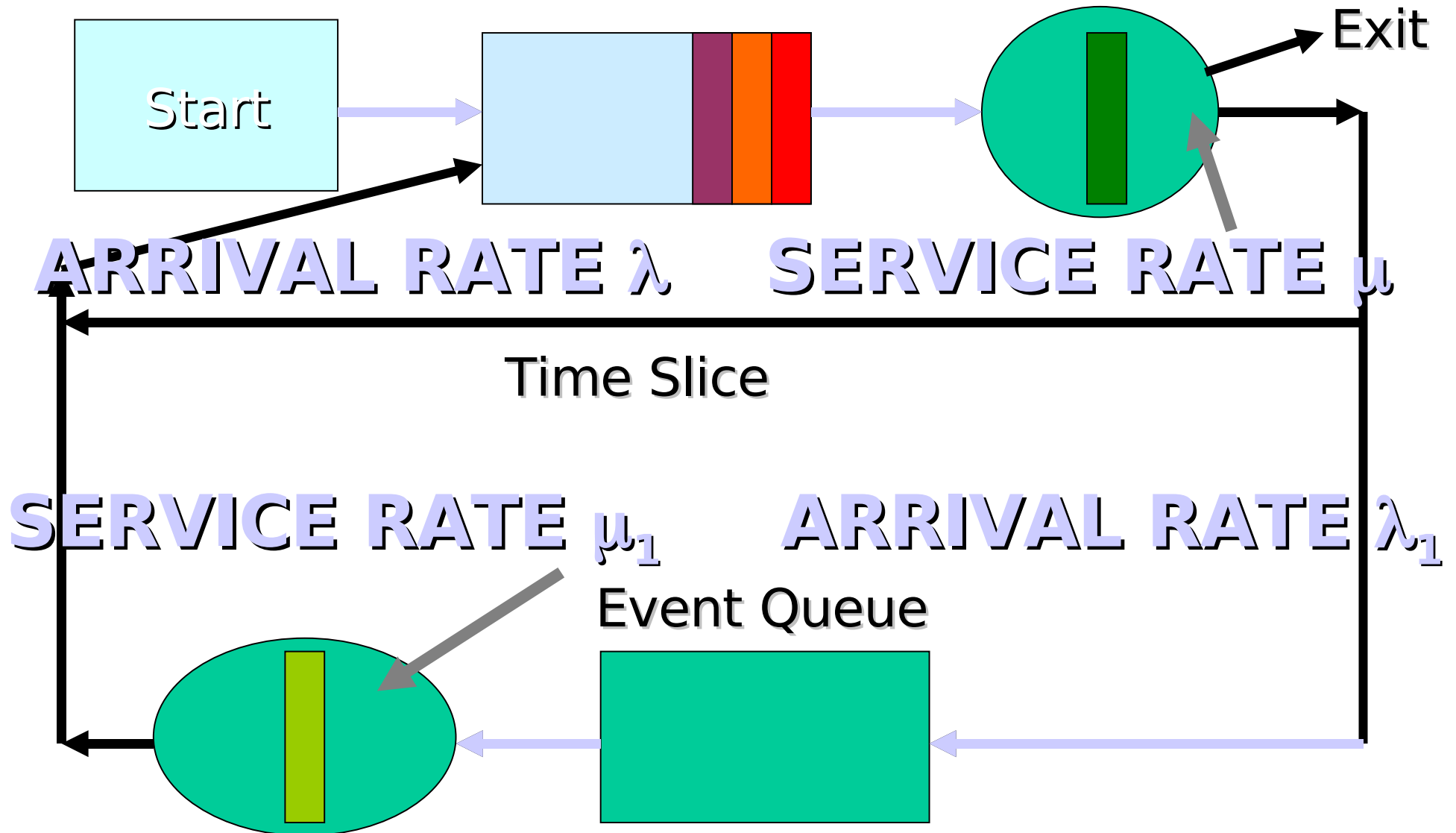
Analysis of Queueing Behavior

Probability n customers arrive in time interval t is:

$$e^{-\lambda t} (\lambda t)^n / n!$$

Assume random service times (also Poisson): μ
constant service rate (customers per unit time)

Queuing Diagram for Processes



Useful Facts From Queuing Theory

W_q = mean time a customer spends in the queue

λ = arrival rate

$L_q = \lambda W_q$ number of customers in queue

W = mean time a customer spends in the system

$L = \lambda W$ (Little's theorem) number of customers in the system

In words – average length of queue is arrival rate times average waiting time

Analysis of Single Server Queue

Server Utilization: $\rho = \lambda/\mu$

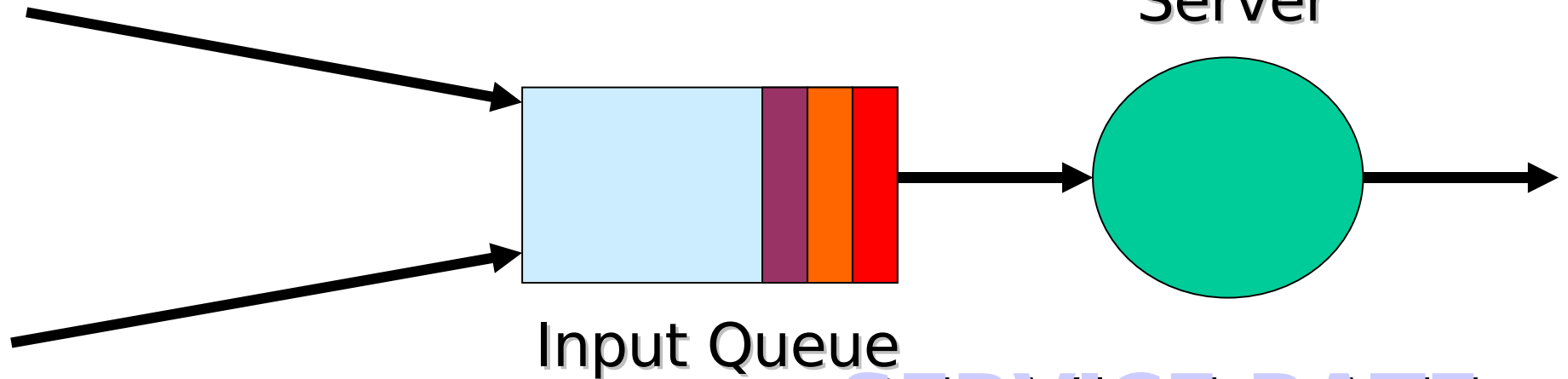
Time in System: $W = 1/(\mu-\lambda)$

Time in Queue: $W_q = \rho/(\mu-\lambda)$

Number in Queue (Little): $L_q = \rho^2/(1-\rho)$

Poisson Arrival & Service Rates Sum

ARRIVAL RATE λ_1



ARRIVAL RATE λ_2

SERVICE RATE μ

$$\lambda = \lambda_1 + \lambda_2$$

LMP1

Queueing

Files Overview

File I/O

Overview

How do you get data in/out of a program?

Unix solution:

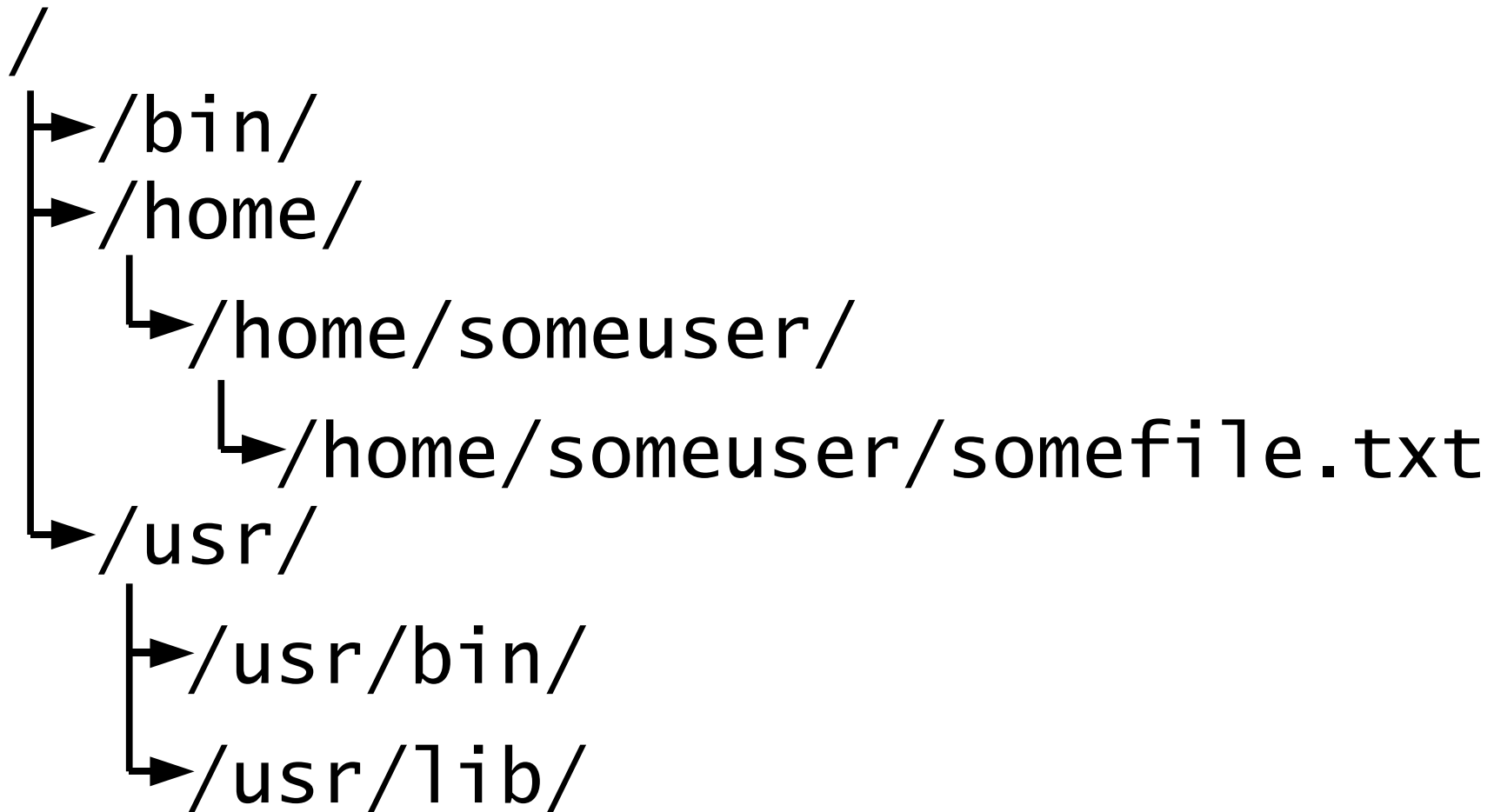
Files.

Many different kinds of I/O are viewed, to some degree, as accessing a file stream:

stdin, stdout, stderr, /dev/audio,

pipes (cat file | grep text), network socket, ...

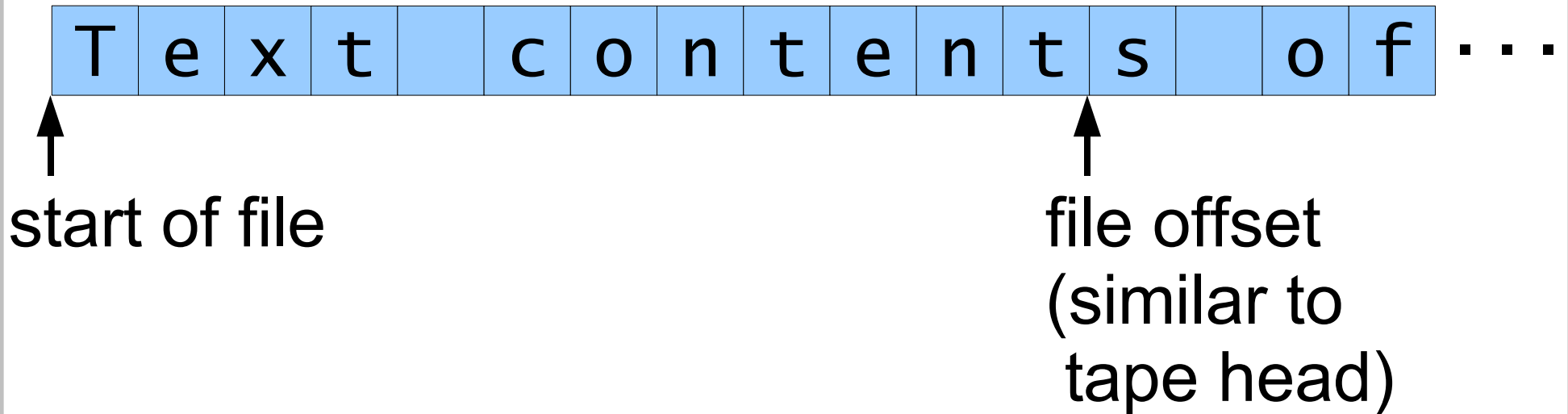
Unix file system tree



Files

A file is structured as a sequence of bytes, modeled after the notion of a *tape*:

“Text contents of some file go here.”



I/O libraries

User process

stdio: fopen, fread, fwrite, fclose, ...
(buffered I/O)

open, read, write, close, select, poll, ...
(direct to kernel I/O)

kernel system call handler

file
I/O

terminal
I/O

pipe
I/O

network
I/O

audio
I/O

Kernel

LMP1

Queueing

Files Overview

File I/O

Buffered I/O: stdio.h

We've previously used stdio.h lightly:

printf(3)

fprintf(3)

etc.

The stdio functions performing *buffering*:

Read a large chunk from a file

When the user requests a few bytes, just read them out of the buffer \Rightarrow lower overhead

Unbuffered kernel I/O

Internally, stdio calls kernel I/O functions:

read(2)

write(2)

open(2)

close(2)

These directly call into the kernel, and only read/write as much as caller requested.

File descriptors

stdio uses a FILE * to denote an open file:

```
FILE *stdin, *stdout, *stderr;
```

The kernel uses int's as *file descriptors*:

0 mean standard input

1 means standard output

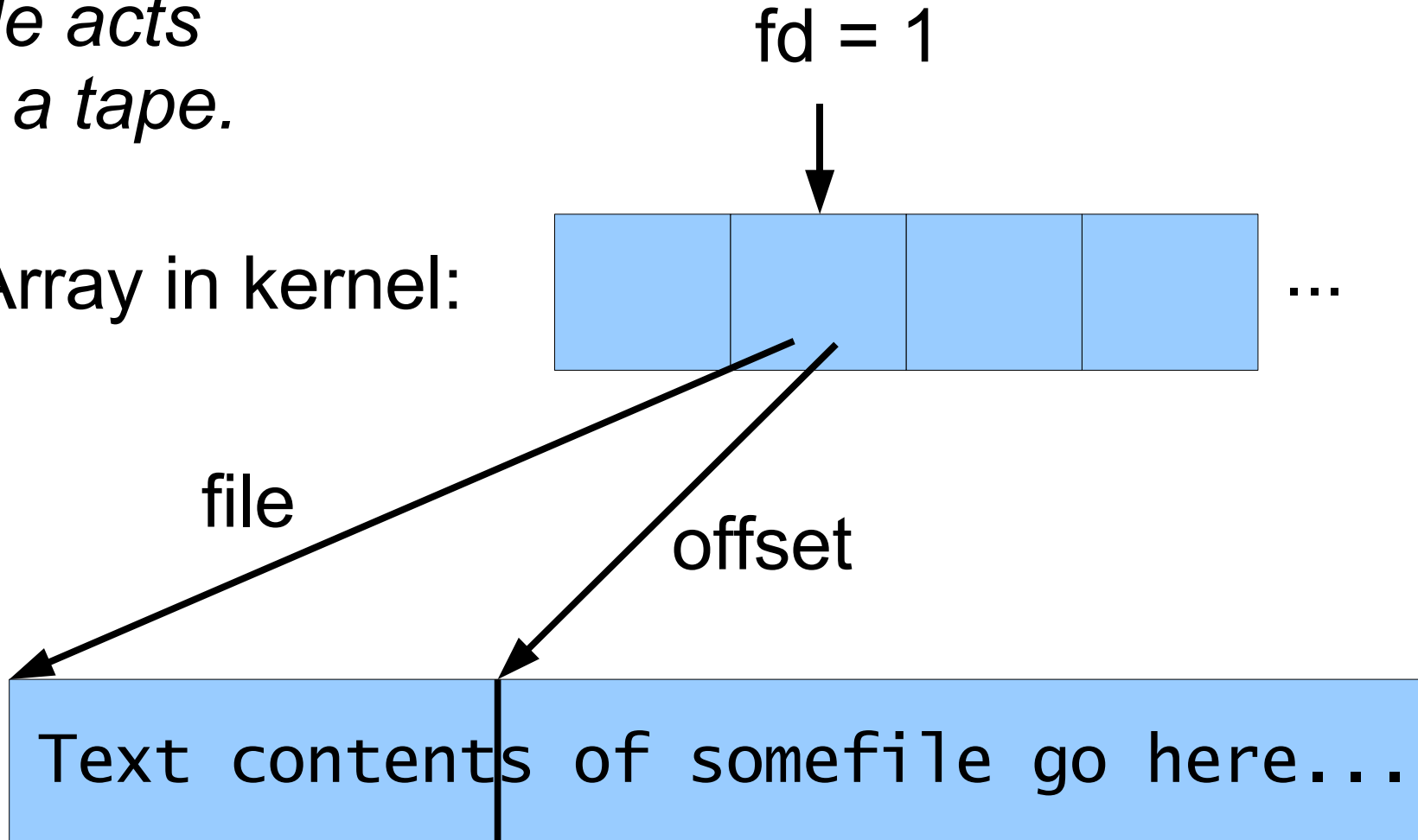
2 means standard error

...

File descriptors

A file acts like a tape.

Array in kernel:



read(2)

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

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

Blocks until data is available.

write(2)

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

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

*Blocks until data can be written
(e.g., pipes block: cat file | grep text).*

open(2)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

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

Flags set access: read, write, append, etc.

close(2)

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

```
int close(int fd);
```

Any file can be closed—even standard input!

Example: show file contents

We'll write a program to print the contents of a file to the terminal (similar to `cat`, but for one file).

```
$ ./show-file somefile.txt  
Text contents of somefile go here.  
$
```

Uses `open(2)`, `read(2)`, `write(2)`, `close(2)`.

lseek(2)

```
#include <sys/types.h>
#include <unistd.h>
```

```
off_t lseek(int fd,
            off_t offset,
            int whence);
```

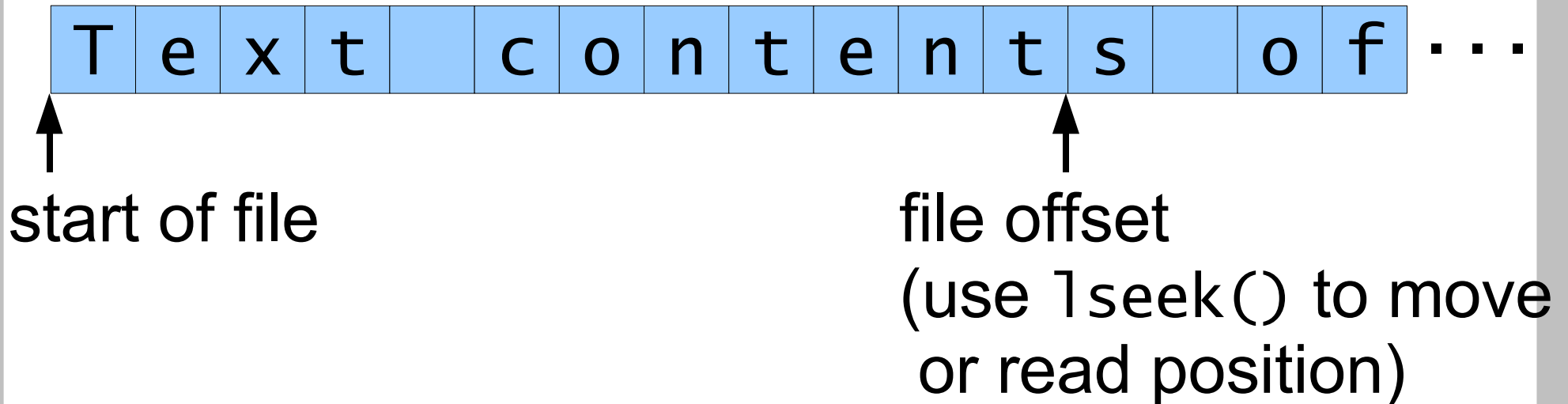
Moves the file offset. whence is one of SEEK_SET, SEEK_CUR, or SEEK_END.

Returns the new offset.

Files

A file is structured as a sequence of bytes, modeled after the notion of a *tape*:

“Text contents of some file go here.”



Example: get byte at offset

Now we'll write a program to print the byte at a given offset in a file, or by default, the file length.

```
$ ./get-byte somefile.txt
35
$ ./get-byte somefile.txt 3
t
$
```

Uses `lseek(2)` plus functions from before.

LMP1

Queueing

Files Overview

File I/O

END

Polling multiple files

What if you have *two* file descriptors open, and want to wait for input on *either one*?

Example:

A web server connected to multiple clients

Unix solutions:

select(2) or poll(2)

select(2)

```
#include <sys/select.h>
```

```
int select(int nfd,  
           fd_set *readfds,  
           fd_set *writefds,  
           fd_set *errorfds,  
           struct timeval *timeout);
```

Waits until read or write will not block, or an error, or timeout expires.

select(2) related macros

Clear out (empty) a set of file descriptors:

```
void FD_ZERO(fd_set *fdset);
```

Add a file descriptor to a set:

```
void FD_SET(int fd, fd_set *fdset);
```

Remove a file descriptor from a set:

```
void FD_CLR(int fd, fd_set *fdset);
```

Check if a file descriptor is in a set:

```
int FD_ISSET(int fd, fd_set *fdset);
```

poll(2)

```
#include <poll.h>
```

```
int poll(struct pollfd *fds,  
         nfds_t        nfd,  
         int           timeout);
```

```
struct pollfd {  
    int    fd;  
    short  events;  
    short  revents;  
};
```

Some poll(2) events

POLLIN

Data is available to read

POLLOUT

Writing now will not block

POLLERR

Error condition

POLLHUP

Hang up (output only)