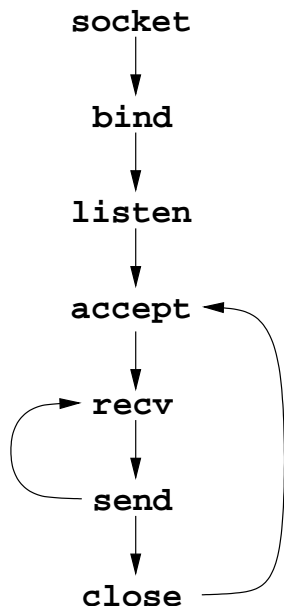


SERVERS

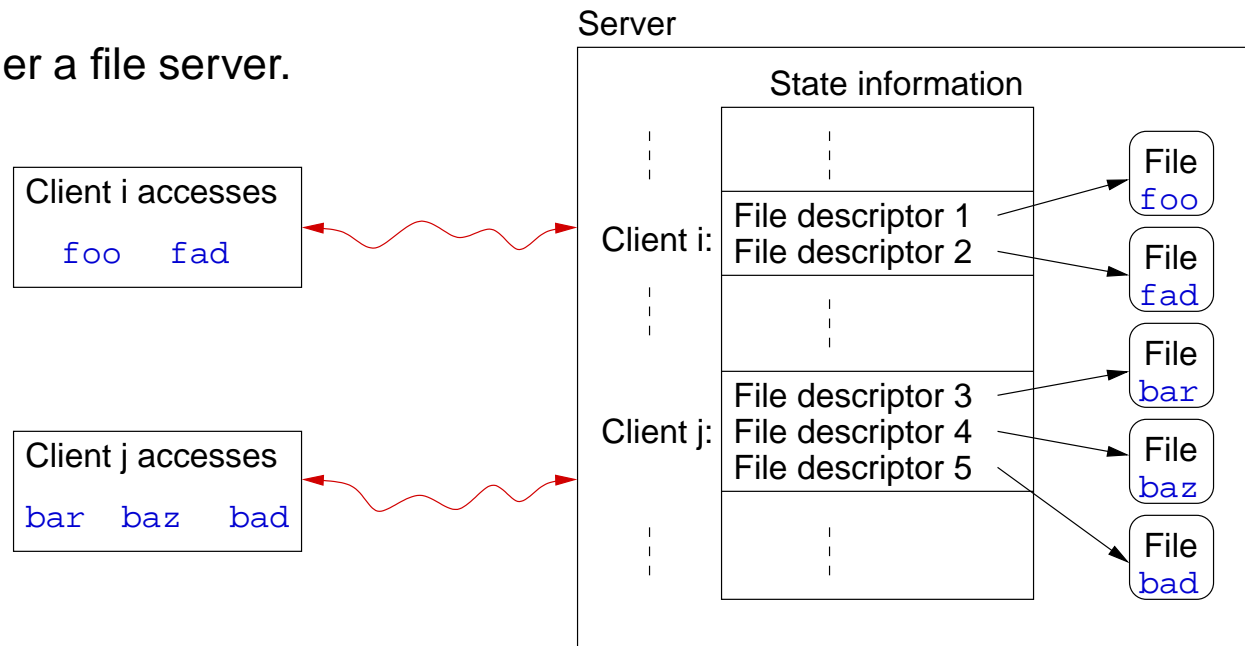


TCP	iterative connection-oriented	concurrent connection-oriented
UDP	iterative connectionless	concurrent connectionless

- We consider TCP servers.
 - point-to-point communication
 - reliable connection establishment and delivery
 - flow-controlled transfer
 - full duplex transfer
 - stream paradigm (no message boundaries)

STATELESSNESS

- While interacting with a client, servers may maintain **state information**.
- Reason: **optimization**.
 - Consider a file server.



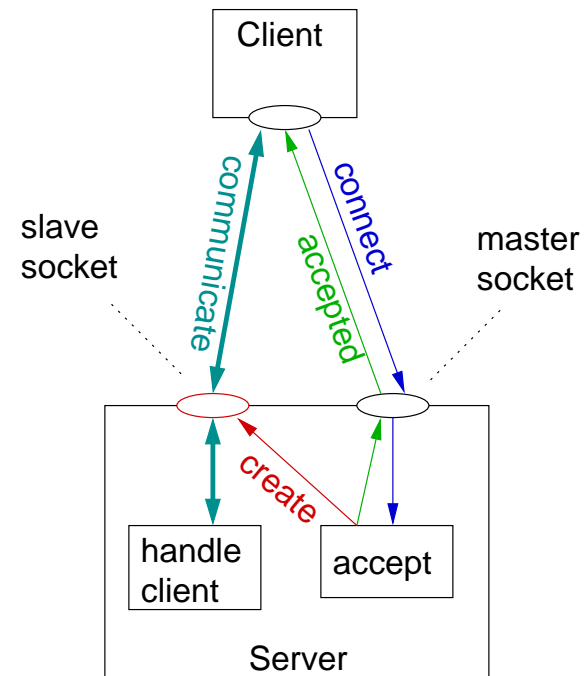
- Useful to keep state information because opening and (especially) seeking may be expensive.

STATELESSNESS (CONT'D)

- Issues in stateful servers:
 - problems with failure of machines or the network
 - * what to do is the server crashes and comes up again? (state information is lost)
 - * what to do with the state information of a client that crashes and comes up again? (state information is not in sync)
 - * what to do with state information for clients that crashed for good? (wastes resources ad infinitum)
 - optimization and managing state information
 - * we could keep files open even when clients no longer use it, in the hope that another client will access the same file
 - * but then we should close files when we open new ones but we run out of space in the descriptor table; which one we close?
 - * we may thus spend more time optimizing the state information than we save by keeping it (**thrashing**)

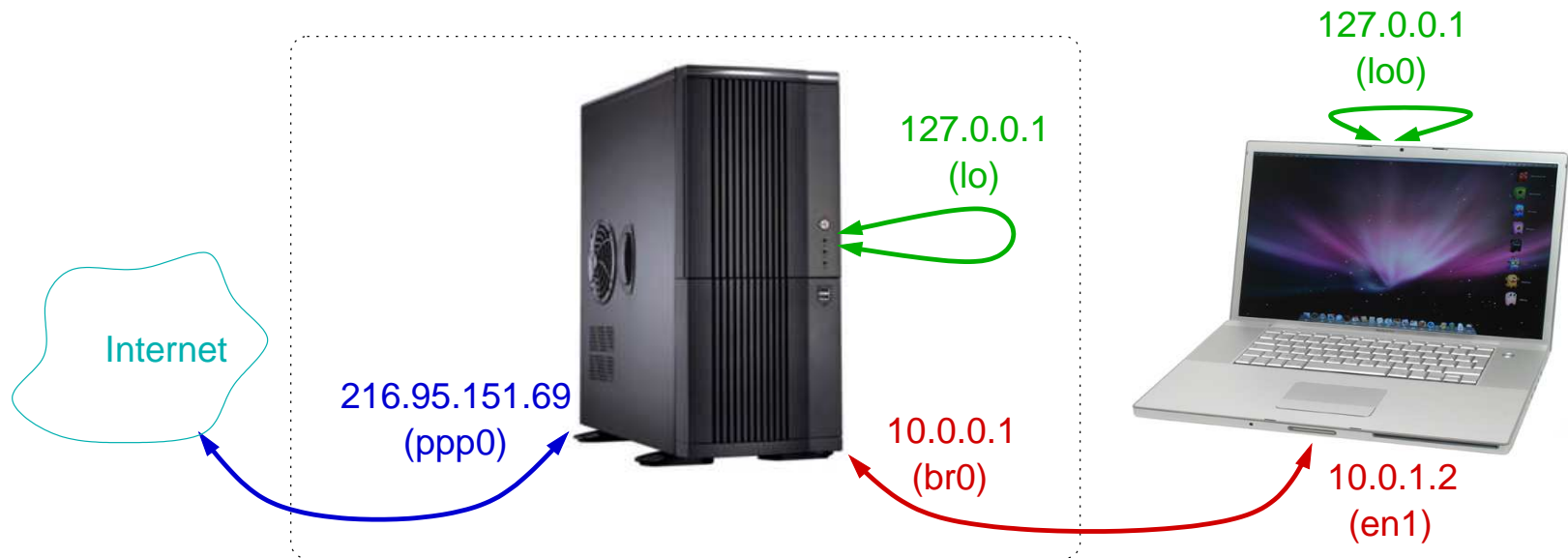
ITERATIVE SERVERS

1. **create** a **master socket**
2. **bind** the socket to a known address (IP address + port number)
3. place the socket in **passive mode**
4. repeat forever:
 - (a) **accept** the next connection request from the socket and **create** a new **slave socket** *s* for the connection.
 - (b) **read** a request from the client
 - (c) **serve** the request and reply to the client
 - (d) if finished with the client, close the socket *s* (civilized servers **shut down *s* first!**); otherwise, repeat from 4b



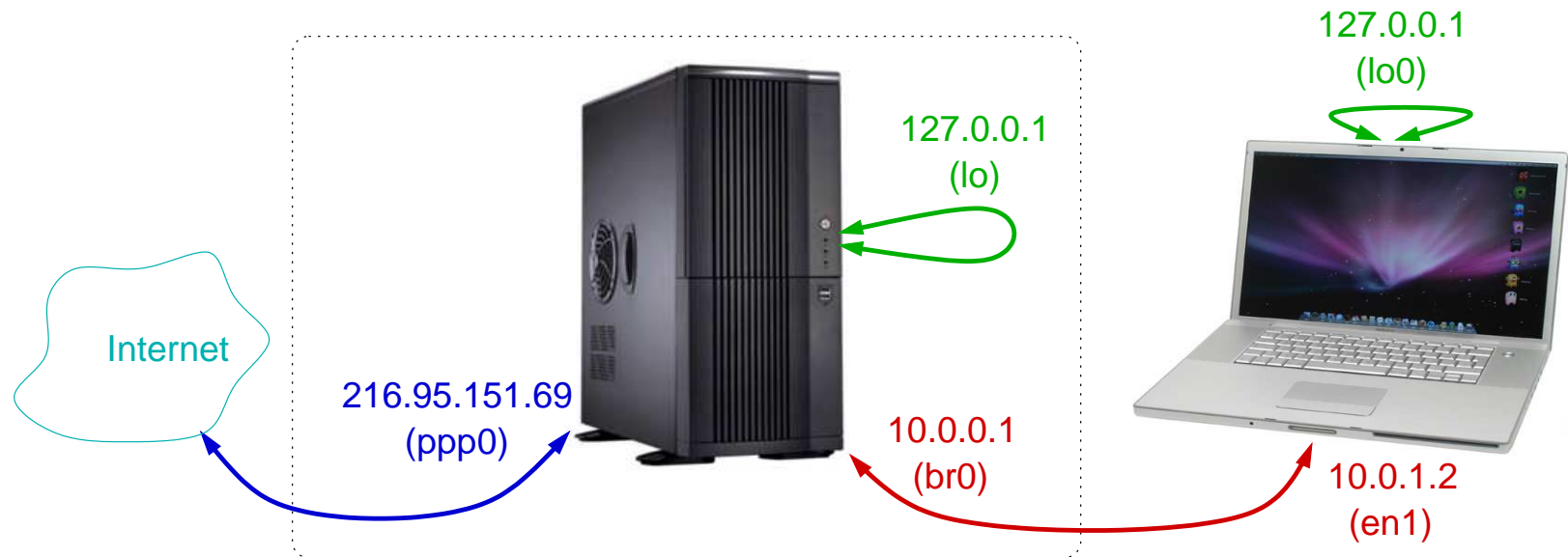
BINDING THE SOCKET

- We specify the IP address and the port number using the structure `sockaddr_in`.
 - Yes, but what address do we provide?



BINDING THE SOCKET

- We specify the IP address and the port number using the structure `sockaddr_in`.
 - Yes, but what address do we provide?



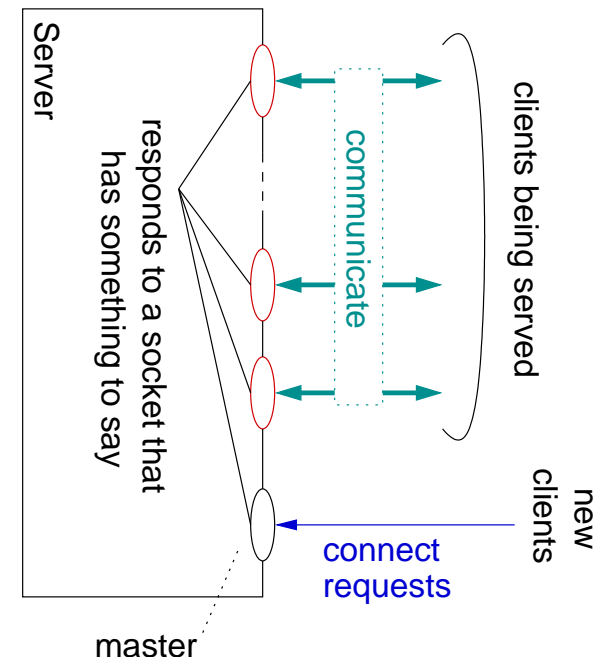
- We can use `INADDR_ANY`
- This denotes a “wildcard” that matches all the IP addresses of the given host.

THE PROBLEM WITH ITERATIVE SERVERS (AND A SOLUTION)

- If two clients connect quasi-simultaneously, one of them will have to wait till the other closes its connection.
 - This could be a loong wait. . .
 - . . . we need some form of concurrency, even if the server itself is iterative (and we **fake** it):

THE PROBLEM WITH ITERATIVE SERVERS (AND A SOLUTION)

- If two clients connect quasi-simultaneously, one of them will have to wait till the other closes its connection.
 - This could be a loong wait. . .
 - . . . we need some form of concurrency, even if the server itself is iterative (and we **fake** it):
1. **create, bind and place in passive mode** the **master socket**
 2. repeat forever:
 - (a) from all the open sockets, **select** a socket s that has data available
 - (b) if s is the master socket, then
 - i. **accept** the next connection request from the socket and **create** a **new** slave socket for the connection.
 - (c) otherwise,
 - i. **read** a request from s
 - ii. **serve** the request and reply
 - iii. if finished with the corresponding client, **close** s



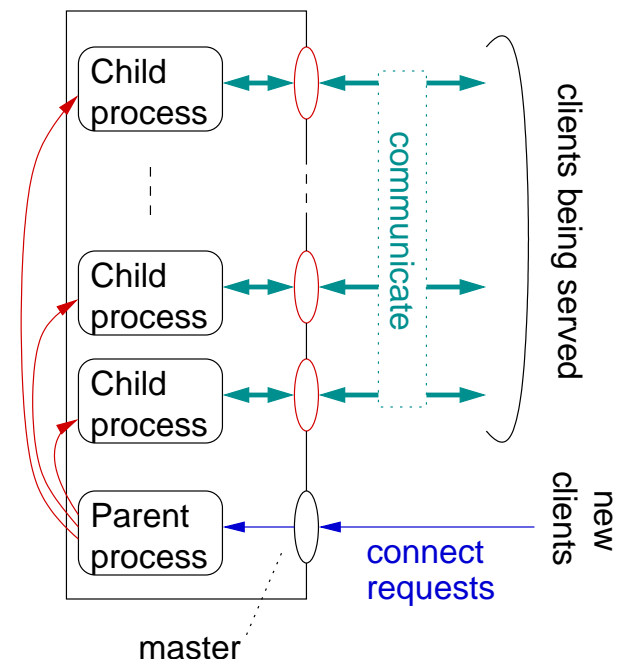
THE PROBLEM WITH APPARENT CONCURRENCY: SERVER DEADLOCK

- Apparent concurrency is prone to **server deadlock**.
 - deadlock happens when a program has to wait for an event to proceed further, but that event never happens.
- Assume that a misbehaved client connects but sends no request.
 - The server calls `recv` or `read` but it never receives data.
 - Thus, it never polls the other sockets again, and the whole server becomes deadlocked.
- True concurrency solves this, as the only locked process is the process serving the misbehaved client.
- Notice however that this is not panaceum for the deadlock problem (so be aware of it even if you write a concurrent server).
 - E.g., a malicious client can send requests but never read the responses.
 - Eventually, the TCP buffers will fill up.
 - Since TCP provides flow control, no new data is transmitted.
- In general, **when the process does a system call and the system cannot satisfy the request, that process will block.**

(REALLY) CONCURRENT SERVERS

- Apparent concurrency is certainly possible, but hairy.
- But then we do not need to fake concurrency since it is offered by the system anyway.

1. **create, bind and place in passive mode** the **master socket**
2. repeat forever:
 - (a) **accept** the next connection request from the socket and **create** a new **slave socket s** for the connection.
 - (b) **fork**
 - (c) if children process then
 - i. **close master socket**
 - ii. **read** a request from the client
 - iii. **serve** the request and reply
 - iv. if finished with the client, close s and terminate; otherwise, repeat from 2(c)ii
 - (d) otherwise (i.e., if parent process),
 - i. **close slave socket**



CONCURRENT SERVERS (CONT'D)

Iterative server:

```
loop
| listen for clients
| if a client requests connection then
|   handle client
|   terminate
| forever
```

Concurrent server:

```
loop
| listen for clients
| if a client requests connection then
|   fork
|   □
|   if child process then
|     handle client
|     terminate
|   □
| forever
```

- Issue: If we use `fork` to create new processes, they will not terminate completely (“zombie processes”).
 - In a server, it is crucial to clean them up.
 - A zombie process dies for good when its parent executes `waitpid` on them.
 - When a child process terminates, it sends a `SIGCHLD` signal to its parent.
 - Ergo, we can create a function `zombie_reaper` that fires up whenever a `SIGCHLD` signal is received:

We then put before the call to `fork`:

```
signal(SIGCHLD, zombie_reaper);
```

```
void zombie_reaper (int sig) {
    int status;
    while (waitpid(-1, &status,
                  WNOHANG) >= 0);
}
```

PROCESSES THAT ACCESS FILES

Scenario:

- We have a concurrent server that accesses a file f .
- A server process wants to write a continuous block of data into f .
- It is quite possible that two processes write to f at the same time, ending up with corrupted data.

Solution: A process that wants to write to the file acquires first a (POSIX) **lock**: With a file descriptor fd , we do: `lockf (fd , F_LOCK , 0) ;`

- If a file is already locked, `lockf (fd , F_LOCK , 0)` blocks (or returns failure, see the `O_NONBLOCK` flag of `open`).
- It is also a good idea to see whether a file is locked: `lockf (fd , F_TEST , 0) ;` (returns -1 and `errno` is set to `EAGAIN`).
- If a file is locked by other process, `write` will block until the lock is released (**beware of deadlocks!**).
- To unlock the file: `lockf (fd , F_ULOCK , 0) ;`

FILE ACCESS (CONT'D)

- The lock created by `lockf` is **mandatory, and enforced by the kernel**.
 - The lock is however **only for writing**. A process can read from a locked file without any problem.
 - You can lock **portions** of files, by giving a non-zero third argument (the offset) to `lockf`.
- To obtain more flexible locks, you should use `fcntl`

```
struct flock lock;
fcntl(fd,F_GETLK,&lock); // get the lock status in l_type (below), or
fcntl(fd,F_SETLK,&lock); // tries to lock, returns -1 on insuccess, or
fcntl(fd,F_SETLKW,&lock); // tries to lock, wait as long as necessary
                          // (deadlock possible!)
```

- The `flock` structure has the following fields:

<code>l_type</code>	type of lock: <code>F_RDLCK</code> , <code>F_WRLCK</code> , or <code>F_UNLCK</code> (set by <code>F_GETLK</code>)
<code>l_whence</code>	where <code>l_start</code> is relative to (like third argument of <code>lseek</code>)
<code>l_start</code>	offset where the lock begins
<code>l_len</code>	size of the locked area (zero means until EOF)
<code>l_pid</code>	process holding the lock

INTERPROCESS COMMUNICATION

- Processes can send **signals** to each other.
 - A signal is just an `int`.
 - It is sent automatically by the system in some cases (e.g., `SIGCHLD` to the parent when the child terminates).
 - But you can also send signals from your code (the meaning of most signals is defined by the system, but `SIGUSR1` and `SIGUSR2` are user-defined).
 - For a description of available signals, see `man -S7 signal`
- To send a signal, use the function `int kill(pid_t pid, int sig);` (see `man -S2 kill`) where `pid` is the process id of some process, or
 - `0` to send the signal to all the processes in the process group
 - `-1` to send the signal to all the processes **except** the first one (`init`)
- To receive a signal, you should establish a **signal handler** in your program by using the function `signal` (see `man -S2 signal`).
 - The signal handler fires **asynchronously** once for each received signal

WHEN SIGNALS ARE NOT ENOUGH

- How do we send data from process to process?

WHEN SIGNALS ARE NOT ENOUGH

- How do we send data from process to process?
 - Using the filesystem (slower)
 - Using `pipes`

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- `filedes[1]` is for writing stuff...
- ...which can then be retrieved from `filedes[0]`

PIPES AND THE UNIX PHILOSOPHY

- Pipes and fork/execve are the most important tools in UNIX
- They allow the implementation of a big chunk of the **UNIX philosophy** (Doug McIlroy et. al.):
 - **Write programs that do one thing and do it well**
 - * To do a new job, build afresh rather than complicate old programs by adding new features
 - * Each time we do something else we do it in a separate process. We can do all of this because **fork-ing is cheap**
 - **Write programs to work together**
 - * Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
 - * We thus couple two programs using **pipes**
 - * Write programs to handle text streams, because this is a universal interface.

THE UNIX PHILOSOPHY (CONT'D)

- Rule of Modularity: Write simple parts connected by clean interfaces
- Rule of Clarity: Clarity is better than cleverness
- Rule of Composition: Design programs to be connected to other programs
- Rule of Separation: Separate policy from mechanism; separate interfaces from engines
- Rule of Simplicity: Design for simplicity; add complexity only where you must
- Rule of Parsimony: Write a big program only when it is clear that nothing else will do
- Rule of Transparency: Design for visibility to make inspection and debugging easier
- Rule of Robustness: Robustness is the child of transparency and simplicity
- Rule of Representation: Fold knowledge into data so program logic can be stupid and robust
- Rule of Least Surprise: In interface design, always do the least surprising thing
- Rule of Silence: When a program has nothing surprising to say, it should say nothing
- Rule of Repair: When you must fail, fail noisily and as soon as possible
- Rule of Economy: Programmer time is expensive; conserve it in preference to machine time
- Rule of Generation: Avoid hand-hacking; write programs to write programs when you can
- Rule of Optimization: Prototype before polishing. Get it working before you optimize it
- Rule of Diversity: Distrust all claims for “one true way”
- Rule of Extensibility: Design for the future, because it will be here sooner than you think

CRITICAL REGIONS

- In a multi-process setting, it is often the case that some piece of code must be executed by one processor at a time.
 - Such a piece of code is called **critical region**.
- One way of mutual exclusion in critical regions is the use of **lock files**:
 - before entering a critical region, a process locks a given file
 - if the file is locked, the process blocks until the lock is released
 - once the critical region is exited, the process releases the lock

ENTERING A CRITICAL REGION

```
int enter_critical (int fd) {
    struct flock lock_info;
    lock_info.l_type = F_WRLCK;
    lock_info.l_whence = SEEK_SET;
    lock_info.l_start = lock_info.l_len = 0;
    int status;

    while ( (status = fcntl(fd,F_SETLKW,&lock_info)) == -1 &&
            errno == EINTR ) {
        // reconstruct lock_info:
        lock_info.l_type = F_WRLCK;
        lock_info.l_whence = SEEK_SET;
        lock_info.l_start = lock_info.l_len = 0;
    }
    if (status == -1)
        perror("enter_critical");
    return status;
}
```

EXITING A CRITICAL REGION

```
int exit_critical (int fd) {
    struct flock lock_info;
    lock_info.l_type = F_UNLCK;
    lock_info.l_whence = SEEK_SET;
    lock_info.l_start = lock_info.l_len = 0;
    int status;

    while ( (status = fcntl(fd,F_SETLKW,&lock_info)) == -1 &&
            errno == EINTR ) {
        // reconstruct lock_info:
        lock_info.l_type = F_UNLCK;
        lock_info.l_whence = SEEK_SET;
        lock_info.l_start = lock_info.l_len = 0;
    }
    if (status == -1)
        perror("exit_critical");
    return status;
}
```

CRITICAL REGIONS (CONT'D)

```
int main (int argc, char** argv) {
    char lock1name[256], lock2name[256];
    snprintf(lock1name, 255, "/tmp/lock-%d-1", getpid());
    snprintf(lock2name, 255, "/tmp/lock-%d-2", getpid());
    int lock1 = open(lock1name, O_WRONLY | O_CREAT | O_APPEND, S_IRWXU | S_IRWXG | S_IRWXO);
    int lock2 = open(lock2name, O_WRONLY | O_CREAT | O_APPEND, S_IRWXU | S_IRWXG | S_IRWXO);

    if (lock1 == -1 || lock2 == -1) {
        perror("Cannot create locks");
        return 1;
    }

    // Do something involving two critical regions...

    // clean up
    close(lock1);
    close(lock2);
    unlink(lock1name);
    unlink(lock2name);
}
```

EXAMPLES

```
if(fork() == 0) {
    if(fork() == 0) { // nephew = 'process 1'
        printf("Process 1 enters critical 1 (%d)\n",
            enter_critical(lock1));
        sleep(3);
        printf("Process 1 exits critical 1 (%d)\n",
            exit_critical(lock1));
    }
    else { // child = 'process 2'
        printf("Process 2 enters critical 1 (%d)\n",
            enter_critical(lock1));
        sleep(3);
        printf("Process 2 exits critical 1 (%d)\n",
            exit_critical(lock1));
    }
}
else { // parent = 'process 3'
    printf("Process 3 enters critical 1 (%d)\n",
        enter_critical(lock1));
    sleep(3);
    printf("Process 3 exits critical 1 (%d)\n",
        exit_critical(lock1));
}
```

Process 3 enters
critical 1 (0).
Process 3 exits
critical 1 (0).
Process 1 enters
critical 1 (0).
Process 1 exits
critical 1 (0).
Process 2 enters
critical 1 (0).
Process 2 exits
critical 1 (0).

EXAMPLES (CONT'D)

```
if(fork() == 0) { // child = 'process 1'
    printf("Process 1 enters critical 1 (%d)\n",
           enter_critical(lock1));
    sleep(1);
    printf("Process 1 enters critical 2 (%d)\n",
           enter_critical(lock2));
    sleep(3);
    printf("Process 1 exits critical 2 (%d)\n",
           exit_critical(lock2));
    printf("Process 1 exits critical 1 (%d)\n",
           exit_critical(lock1));
}
else { // parent = 'process 2'
    printf("Process 2 enters critical 2 (%d)\n",
           enter_critical(lock2));
    sleep(1);
    printf("Process 2 enters critical 1 (%d)\n",
           enter_critical(lock1));
    sleep(3);
    printf("Process 2 exits critical 2 (%d)\n",
           exit_critical(lock2));
    printf("Process 2 exits critical 1 (%d)\n",
           exit_critical(lock1));
}
```

EXAMPLES (CONT'D)

```
if(fork() == 0) { // child = 'process 1'
    printf("Process 1 enters critical 1 (%d)\n",
           enter_critical(lock1));
    sleep(1);
    printf("Process 1 enters critical 2 (%d)\n",
           enter_critical(lock2));
    sleep(3);
    printf("Process 1 exits critical 2 (%d)\n",
           exit_critical(lock2));
    printf("Process 1 exits critical 1 (%d)\n",
           exit_critical(lock1));
}
else { // parent = 'process 2'
    printf("Process 2 enters critical 2 (%d)\n",
           enter_critical(lock2));
    sleep(1);
    printf("Process 2 enters critical 1 (%d)\n",
           enter_critical(lock1));
    sleep(3);
    printf("Process 2 exits critical 2 (%d)\n",
           exit_critical(lock2));
    printf("Process 2 exits critical 1 (%d)\n",
           exit_critical(lock1));
}
```

```
Process 2 enters
critical 2 (0).
Process 1 enters
critical 1 (0).
enter_critical:
Resource deadlock
avoided
Process 2 enters
critical 1 (-1).
Process 1 enters
critical 2 (0).
Process 2 exits
critical 2 (0).
Process 2 exits
critical 1 (0).
Process 1 exits
critical 2 (0).
Process 1 exits
critical 1 (0).
```

FILE LOCKING IMPLEMENTATION

- The theory is nice by the practice is messy...
- On Linux `lockf(3)` is just a wrapper for `fcntl(2)`; on other systems this might not be the case (so don't mix and match)
- On BSD mandatory and advisory locking are aware of each other (so you can mix and match at will), other systems might see things differently
- A lock is mandatory iff the file is **marked** as a candidate for mandatory locking by **setting the group-id bit in its file mode but removing the group-execute bit** — an otherwise meaningless combination chosen so as not to break existing user programs
- Solaris, HP-UX reject all calls to `open()` for a file on which another process has outstanding mandatory locks — direct contravention of the System V standard (but not POSIX), which states that only calls to `open()` with the `O_TRUNC` flag set should be rejected; Linux follow the System V standard
- Race conditions can still appear
- Quote from the Linux kernel: **I'm afraid that this is such an esoteric area that the semantics described below are just as valid as any others, so long as the main points seem to agree**

FILE LOCKING IMPLEMENTATION (LINUX)

- Mandatory locks can only be applied via the `fcntl()/lockf()` locking interface = the System V/POSIX interface. BSD style locks using `flock()` never result in a mandatory lock.
- If a process has locked a region of a file with a mandatory read lock, then other processes are permitted to read from that region. If any of these processes attempts to write to the region it will block until the lock is released, unless the process has opened the file with the `O_NONBLOCK` flag in which case the system call will return immediately with the error status `EAGAIN`.
- If a process has locked a region of a file with a mandatory write lock, all attempts to read or write to that region block until the lock is released, unless a process has opened the file with the `O_NONBLOCK` flag in which case the system call will return immediately with the error status `EAGAIN`.
- Calls to `open()` with `O_TRUNC`, or to `creat()`, on an existing file that has any mandatory locks owned by other processes will be rejected with the error status `EAGAIN`.
- Not even root can override a mandatory lock, so runaway processes can wreak havoc if they lock crucial files!
 - Way around: remove the setgid bit (hard to do on a hung system)

THE PERILS OF POSIX LOCKS

- POSIX locks (`lockf`) are mandatory and enforced by the kernel
- ...but there are times in which they do not do what you want them to do:
 - Contrary to the popular belief, the following program will **not** block on the lock:

```
int fd = open("aaa", O_RDWR);
if (fork() == 0) {
    lockf(fd, F_LOCK, 0);
    printf("child locked \"aaa\"..."); sleep(5); printf(" done\n");
}
else {
    sleep(1);
    write(fd, "stuff", 5);
    printf("parent wrote into \"aaa\"\n");
}
```

- POSIX locks are established at **descriptor level** and are thus **inherited** by child processes
- However, attempting to place a lock on an already locked descriptor will positively block

ADVISORY LOCKS

- Advisory locks (`flock`) look almost the same... but are only advisory
- That is, any program may choose not to take them into account
- Avoiding `flock` in favour of POSIX locks is probably a good idea
 - but then do take into consideration the issues related to your program locking a crucial file and then hanging
- Advisory locks are by the way inherited by a child process but **not** across `execve`