7. Communication and Synchronization

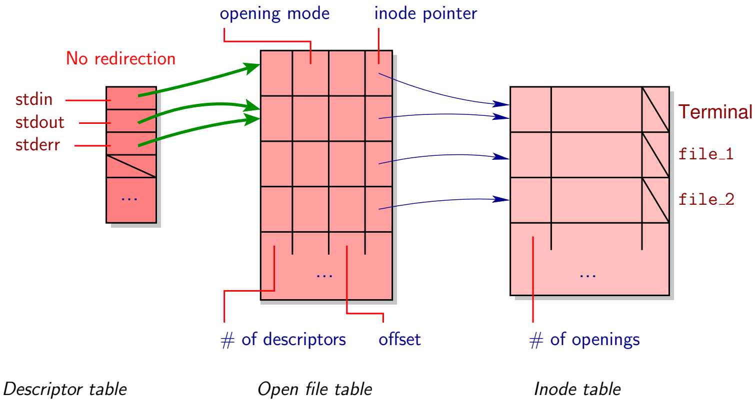
Outline

- I/O redirection
- Inter-process communication and synchronization
  - FIFO (pipe)
  - Advanced event flow (signals)
  - Message queues
  - Shared memory
    - Mapping I/O to memory
    - Shared memory segments
  - Mutual exclusion
    - Semaphores
    - File locking
- Programmer interface
  - Main system calls
  - Examples

I/O Redirection

Example

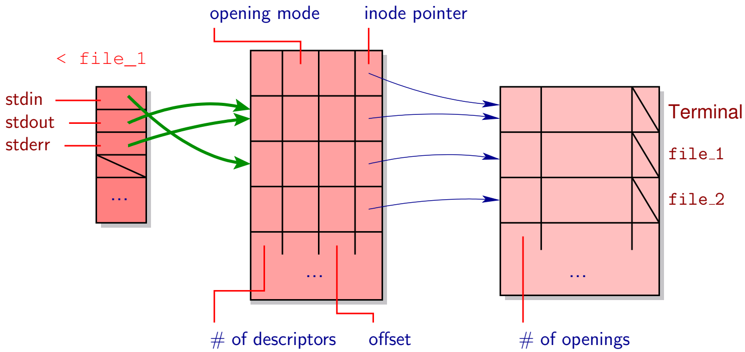
No redirection



I/O Redirection

Example

Standard input redirection

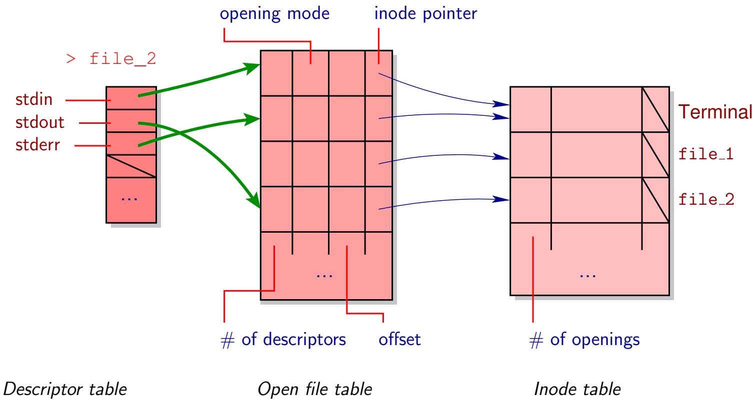


Descriptor table *Open file table* *Inode table*

I/O Redirection

Example

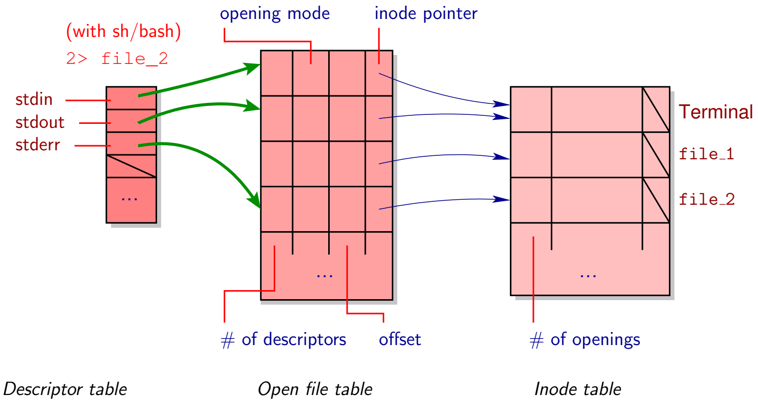
Standard output redirection



I/O Redirection

Example

Standard error redirection



I/O System Call: `dup()`/`dup2()`

Duplicate a File Descriptor

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Return Value

- On success, `dup()` and `dup2()` return a (non-negative) *file descriptor*, which is a copy of `oldfd`
  - For `dup()`, it is the process's *lowest-numbered descriptor not currently open*
  - `dup2()` uses `newfd()` instead, closing it before if necessary
  - Clears the flags of the new descriptor (see `fcntl()`)
  - Both descriptors share one single open file (i.e., one offset for `lseek()`, etc.)
- Returns -1 on error

Error Conditions

- An original `errno` code

  `EMFILE`: too many file descriptors for the process

Redirection Example

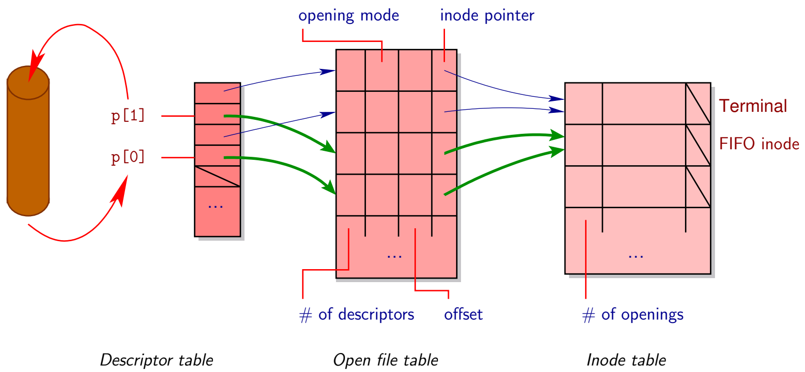
```
$ command > file_1   // Redirect stdout to file_1

{
  close(1);
  open("file_1", O_WRONLY | O_CREAT, 0777);
}
```

```
$ command 2>&1       // Redirect stderr to stdout

{
  close(2);
  dup(1);
}
```

FIFO (Pipe)

Principles

- Channel to stream data among processes
  - Data traverses the pipe first-in (write) first-out (read)
  - Blocking read and write (bounded capacity)
  - Illegal to write in a pipe without reader
  - Reading in a pipe without writer "simulates end of file"



I/O System Call: `pipe()`

Create a Pipe

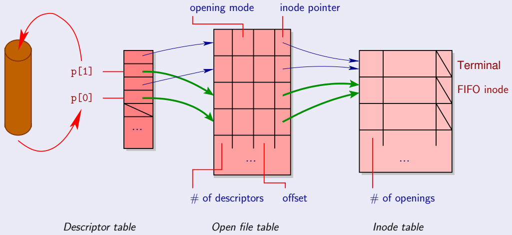
```
#include <unistd.h>

int pipe(int p[2]);
```

Description

- Creates a pipe and stores a pair of file descriptors into `p`
  - `p[0]` for reading (`O_RDONLY`)
  - `p[1]` for writing (`O_WRONLY`)
- Returns $\mathbf{0}$ on success, $-\mathbf{1}$ if an error occurred

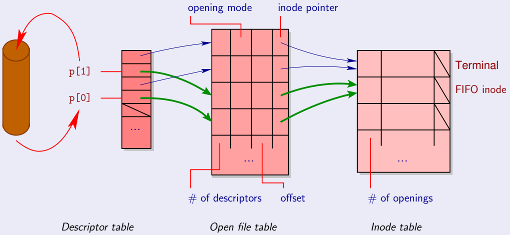
FIFOs and I/O Redirection

- Question: implement
  `$ ls | more`



FIFOs and I/O Redirection

- Question: implement
  `$ ls | more`
- Solution
  - `pipe(p)`
  - `fork()`
  - Process to become `ls`
    - `close(1)`
    - `dup(p[1])`
    - `execve("ls", ...)`
  - Process to become `more`
    - `close(0)`
    - `dup(p[0])`
    - `execve("more", ...)`



FIFO Special Files

Named Pipe

- Special file created with `mkfifo()` (front-end to `mknod()`)
  See also `mkfifo` command
- Does not store anything on the file system (beyond its inode)
  Data is stored and forwarded in memory (like an unnamed pipe)
- Supports a rendez-vous protocol
  - Open for reading: blocks until another process opens for writing
  - Open for writing: if no reader, fails with error `ENXIO`
- Disabled when opening in `O_NONBLOCK` mode

Pipe I/O

Writing to a Pipe

- Writing to a pipe without readers delivers of `SIGPIPE`
  - Causes termination by default
  - Otherwise causes `write()` to fail with error `EINTR`
- `PIPE_BUF` is a constant $\geqslant$ **512** (**4096** on Linux)
- Writing **n** bytes in blocking mode
  - **n** $\leqslant$ `PIPE_BUF`: atomic success (**n** bytes written), block if not enough space
  - **n** $>$ `PIPE_BUF`: non-atomic (may be interleaved with other), blocks until **n** bytes have been written
- Writing **n** bytes in non-blocking mode (`O_NONBLOCK`)
  - **n** $\leqslant$ `PIPE_BUF`: atomic success (**n** bytes written), or fails with `EAGAIN`
  - **n** $>$ `PIPE_BUF`: if the pipe is full, fails with `EAGAIN`; otherwise a partial write may occur
- Reading from a pipe without writer returns **0** (end of file)
- Like ordinary files, data sent to a pipe is unstructured: it does not retain "boundaries" between calls to `write` (unlike IPC message queues)

Advanced Synchronization With Signals

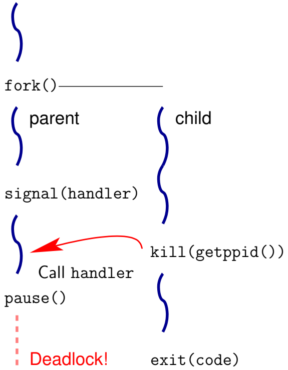
Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?

Advanced Synchronization With Signals

Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?



Advanced Synchronization With Signals

Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?
    Asynchronous signal delivery
    $\longrightarrow$ Possible deadlock
    $\longrightarrow$ Hard to fix the bug

Advanced Synchronization With Signals

Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?
    Asynchronous signal delivery
    $\longrightarrow$ Possible deadlock
    $\longrightarrow$ Hard to fix the bug
- Solution: atomic (un)masking (a.k.a. (un)blocking) and suspension

Advanced Synchronization With Signals

Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?
    Asynchronous signal delivery
    $\longrightarrow$ Possible deadlock
    $\longrightarrow$ Hard to fix the bug
- Solution: atomic (un)masking (a.k.a. (un)blocking) and suspension
- Lessons learned
  - Difficult to tame low-level concurrency mechanisms
  - Look for *deterministic* synchronization/communication primitives (enforce functional semantics)

System Call: sigaction()

POSIX Signal Handling

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
                          struct sigaction *oldact);
```

Description

- Examine and change the action taken by a process on signal delivery
- If `act` is not `NULL`, it is the new action for signal `signum`
- If `oldact` is not `NULL`, store the current action into the `struct sigaction` pointed to by `oldact`
- Return **0** on success, **−1** on error

System Call: sigaction()

POSIX Signal Handling

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
                          struct sigaction *oldact);
```

Error Conditions

- Typical error code

  EINVAL: an invalid signal was specified, or attempting to change the action for SIGKILL or SIGSTOP
  Calling sigaction() with NULL second and third arguments and checking for the EINVAL error allows to check whether a given signal is supported on a given platform

System Call: sigaction()

POSIX Signal Action Structure

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t*, void*);
  sigset_t sa_mask;
  int sa_flags;
}
```

Description

- sa_handler: same function pointer as the argument of signal() (it may also be set to SIG_DFL or SIG_IGN)
- sa_sigaction: handler with information about the context of signal delivery (excusive with sa_handler)
- sa_mask: mask of blocked signals when executing the signal handler
- sa_flags: bitwise or of handler behavior options

System Call: `sigaction()`

POSIX Signal Action Structure

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t*, void*);
  sigset_t sa_mask;
  int sa_flags;
}
```

`SA_NOCLDSTOP`: if `signum` is `SIGCHLD`, no notification when child processes stop (`SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`) or resume (`SIGCONT`)

`SA_NOCLDWAIT`: if `signum` is `SIGCHLD`, "leave children *unattended*", i.e., do not transform terminating children processes into zombies

`SA_SIGINFO`: use `sa_sigaction` field *instead* of `sa_handler`

- `siginfo_t` parameter carries signal delivery context
- `$ man 2 sigaction` for (lengthy) details

A few others: reset handler after action, restart interrupted system calls, authorize nesting of identical signals, etc.

The sigsetops Family of Signal-Set Operations

```
$ man 3 sigsetops

#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

Description

- Respectively empty all signals, fill with all signals, add a signal, remove a signal, and test whether a signal belong to the POSIX `sigset_t` pointed to by `set`
- The first four return $\mathbf{0}$ on success and $-\mathbf{1}$ on error
- `sigismember()` returns $\mathbf{1}$ if `signum` is a member of the set, $\mathbf{0}$ if it is not, and $-\mathbf{1}$ on error
- See also the non-portable `sigisemptyset()`, `sigorset()`, `sigandset()`

Simple sigaction Example

```
int count_signal = 0;

void count(int signum) {
  count_signal++;
}

// ...

{
  struct sigaction sa;

  sa.sa_handler = count;                    // Signal handler
  sigemptyset(&sa.sa_mask);                 // Pass field address directly
  sa.sa_flags = 0;
  sigaction(SIGUSR1, &sa, NULL);

  while (true) {
    printf("count_signal = %d\n", count_signal);
    pause();
  }
}
```

System Call: sigprocmask()

Examine and Change Blocked Signals

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Semantics

- If set is not NULL, how describes the behavior of the call

  SIG_BLOCK: blocked $\leftarrow$ blocked $\cup$ *set
  SIG_UNBLOCK: blocked $\leftarrow$ blocked $-$ *set
  SIG_SETMASK: blocked $\leftarrow$ *set

- If oldset is not NULL, store the current mask of blocked signals into the sigset_t pointed to by oldset
- Return **0** on success, $-\mathbf{1}$ on error

System Call: `sigprocmask()`

Examine and Change Blocked Signals

```c
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Remarks

- Unblockable signals: `SIGKILL`, `SIGSTOP`
  (attempts to mask them are silently ignored)
- Use `sigsuspend()` to unmask signals before suspending execution

System Call: `sigpending()`

Examine Pending Signals

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Semantics

- A signal is *pending* if it has been delivered but not yet handled, because it is currently blocked
  (or because the kernel did not yet check for its delivery status)
- Stores the set of pending signals into the `sigset_t` pointed to by `set`
- Return **0** on success, $-\mathbf{1}$ on error

System Call: `sigsuspend()`

Wait For a Signal

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

Semantics

- Perform the two following operations *atomically* w.r.t. signal delivery
  1. Set `mask` as the temporary set of masked signals
  2. Suspend the process until delivery of an *unmasked*, *non-ignored* signal
- When recieving a non-terminating, non-ignored signal, execute its handler *before* restoring the previous set of masked signals and resuming execution
- Always return $-\mathbf{1}$, typically with error code `EINTR`

System Call: `sigsuspend()`

Wait For a Signal

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

Typical Usage

- *Prevent early signal delivery between unmasking and suspension*
  1. Call `sigprocmask()` to disable a set of signals
  2. Perform some critical operation
  3. Call `sigsuspend()` to atomically enable some of them and suspend execution
- Without this atomic operation (i.e., with `signal()` and `pause()`)
  1. A signal may be delivered *between* the installation of the signal handler (the call to `signal()`) and the suspension (the call to `pause()`)
  2. Its handler (installed by `signal()`) may be triggered *before the suspension* (the call to `pause()`)
  3. Handler execution *clears the signal* from the process's pending set
  4. The suspended process deadlocks, waiting for an already-delivered signal
- No (direct) way to avoid this dangerous "race" using ISO C signals

Example With Signals and Memory Management

```
#include <stdio.h>
#include <signal.h>
struct sigaction sa;
void *p;
void catch(int signum) {        // Catch a segmentation violation
  static int save_p == NULL;
  if (save_p == NULL) {
    save_p = p;
    brk(p+1);
  } else {
    printf("Page size: %d\n", p - save_p);
    exit(0);
  }
}
int main(int argc, char *argv[]) {
  sa.sa_handler = catch; sigemptyset(&sa.sa_mask); sa.sa_flags = 0;
  sigaction(SIGSEGV, &sa, NULL);
  p = sbrk(0);
  while (1)
    *p++ = 42;
}

$ page
Page size: 4096
```

IPC: Message Queues

Queueing Mechanism for Structured Messages

- Signals
  - Carry no information beyond their own delivery
  - Cannot be queued
- FIFOs (pipes)
  - Unstructured stream of data
  - No priority mechanism
- Message queues offer a prioritized, loss-less, structured communication channel between processes

  `$ man 7 mq_overview`

Implementation in Linux

- Message queue files are single inodes located in a specific *pseudo-file-system*, mounted under `/dev/mqueue`
- Must link the program with `-lrt` (real-time library)

System Call: mq_open()

Open and Possibly Create a POSIX Message Queue

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int flags);
mqd_t mq_open(const char *name, int flags, mode_t mode,
              struct mq_attr *attr);
```

Description

- Analogous to open(), but not mapped to persistent storage
- Argument name must begin with a "/" and may not contain any other "/"
- Arguments flags and mode allow for a subset of their values for open()
  - flags: O_RDONLY, O_RDWR, O_CREAT, O_EXCL, O_NONBLOCK, but *not* O_APPEND, O_TRUNC, or any other flag
    Note: FD_CLOEXEC flag is set automatically
  - mode: S_IRUSR, S_IWUSR, S_IXUSR, etc.
  - attr: attributes for the queue, *see* mq_getattr()
    Default set of attributes if NULL or not specified
- Returns a message queue descriptor on success, **−1** on error

System Call: mq_getattr() and mq_setattr()

Get/Set Attributes of a POSIX Message Queue

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *mq_attr)
int mq_setattr(mqd_t mqdes, struct mq_attr *mq_newattr,
               struct mq_attr *mq_oldattr)
```

Description

- The mq_attr structure is defined as

```
struct mq_attr {
  long mq_flags;        // Flags: 0 or O_NONBLOCK
  long mq_maxmsg;       // Maximum # of messages in queue
  long mq_msgsize;      // Maximum message size (bytes)
  long mq_curmsgs;      // # of messages currently in queue
};
```

- mq_maxmsg and mq_msgsize cannot be modified
- Returns **0** on success, **−1** on error

System Call: mq_send()

Send a Message To a POSIX Message Queue

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, char *msg_ptr,
            size_t msg_len, unsigned int msg_prio)
```

Description

- Enqueues the message pointed to by msg_ptr of size msg_len into mqdes
- msg_len must be less than or equal to the mq_msgsize attribute of the queue (*see* mq_getattr())
- msg_prio is a non-negative integer specifying message priority
  0 is the lowest priority, and **31** is the highest (portable) priority
- By default, mq_send() blocks when the queue is full (i.e., mq_maxmsg currently in queue)
- Returns **0** on success, **−1** on error

System Call: mq_receive()

Receive a Message From a POSIX Message Queue

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned int *msg_prio)
```

Description

- Removes the oldest message with the highest priority from mqdes
- Stores it into the buffer pointed to by msg_ptr of size msg_len
- msg_len must be greater than or equal to the mq_msgsize attribute of the queue (*see* mq_getattr())
- If msg_prio is not null, use it to store the priority of the received message
- By default, mq_receive() blocks when the queue is empty
- Returns the number of bytes of the received message on success, -1 on error

System Call: mq_close()

Close a POSIX Message Queue Descriptor

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

Description

- Also removes any notification request attached by the calling process to this message queue
- Returns $\mathbf{0}$ on success, $\mathbf{-1}$ on error

System Call: mq_unlink()

Unlink a POSIX Message Queue File

```
#include <mqueue.h>

int mq_close(const char *name);
```

Description

- Message queues have *kernel* persistence
- Similar to unlink()

Other System Calls

- mq_notify(): notify a process with a signal everytime the specified queue receives a message while originally empty
- mq_timedreceive() and mq_timedsend(): receive and send with timeout

Memory and I/O Mapping

Virtual Memory Pages

- *Map* virtual addresses to physical addresses
  - Configure MMU for page translation
  - Support growing/shrinking of virtual memory segments
  - Provide a protection mechanism for memory *pages*
- Implement copy-on-write mechanism (e.g., to support `fork()`)

Memory and I/O Mapping

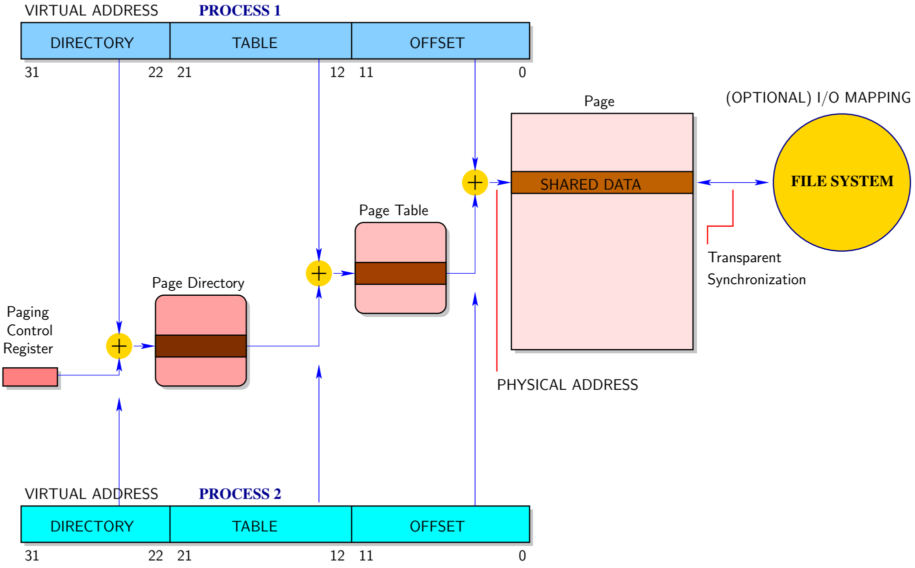
Virtual Memory Pages

- *Map* virtual addresses to physical addresses
  - Configure MMU for page translation
  - Support growing/shrinking of virtual memory segments
  - Provide a protection mechanism for memory *pages*
- Implement copy-on-write mechanism (e.g., to support `fork()`)

I/O to Memory

- *Map* I/O operations to simple memory load/store accesses
- Facilitate sharing of memory pages
  - Use file naming scheme to identify memory regions
  - Same system call to implement private and shared memory allocation

Memory and I/O Mapping



System Call: `mmap()`

Map Files or Devices Into Memory

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

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Semantics

- Allocates `length` bytes from the process virtual memory, starting at the `start` address or any fresh interval of memory if `start` is `NULL`
- Maps to this memory interval the region of a file specified by `fd` and starting at position `offset`
- The `start` address must be a multiple of the virtual memory page size; it is almost always `NULL` in practice (leaving the reponsibility of choosing an address to the kernel)
- Return value
  - Start address of the mapped memory interval on success
  - `MAP_FAILED` on error (i.e., `(void*)-1`)

System Call: mmap()

Map Files or Devices Into Memory

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

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Memory Protection: the prot Argument

- It may be PROT_NONE: access forbiden
- Or it may be built by *or'ing* the following flags

  PROT_EXEC: data in pages may be executed as code
  PROT_READ: pages are readable
  PROT_WRITE: pages are writable

System Call: mmap()

Map Files or Devices Into Memory

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

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Memory Protection: the flags Argument

- Either

  MAP_PRIVATE: create a private, copy-on-write mapping; writes to the region do not affect the mapped file

  MAP_SHARED: share this mapping with all other processes which map this file; writes to the region affect the mapped file

  MAP_ANONYMOUS: mapping not associated to any file (fd and offset are ignored); underlying mechanism for growing/shrinking virtual memory segments (including stack management and malloc())

System Call: mmap()

Map Files or Devices Into Memory

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

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Error Conditions

EACCESS: fd refers to non-regular file or prot incompatible with opening mode or access rights
Note: modes O_WRONLY, O_APPEND are forbidden

ENOMEM: not enough memory

Error Signals

SIGSEGV: violation of memory protection rights

SIGBUS: access to memory region that does not correspond to a legal position in the mapped file

System Call: munmap()

Delete a Memory Mapping for a File or Device

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

int munmap(void *start, size_t length);
```

Semantics

- Deletes the mappings for the specified address and range
- Further accesses will generate invalid memory references
- Remarks
  - start must be multiple of the page size (typically, an address returned by mmap() in the first place)
    Otherwise: generate SIGSEGV
  - All pages containing part of the specified range are unmapped
  - Any pending modification is synchronized to the file
    See also msync()
  - Closing a file descriptor does not unmap the region
- Returns **0** on success, **−1** on error

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question
  How do multiple processes *agree* on a *sharing* a region of *physical memory*?

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question
  How do multiple processes *agree* on a *sharing* a region of *physical memory*?
- *Sharing* is easy: call `mmap()` with `MAP_SHARED` flag
- *Agreeing* is the problem
  Solution: use a *file name* as a meeting point

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question
  How do multiple processes *agree* on a *sharing* a region of *physical memory*?
- *Sharing* is easy: call `mmap()` with `MAP_SHARED` flag
- *Agreeing* is the problem
  Solution: use a *file name* as a meeting point
- Minor problem... one may not want to waste disk space for transient data (not persistent accross system shutdown)
  `MAP_ANONYMOUS` solves this problem... but looses the association between the file and memory region

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question
  How do multiple processes *agree* on a *sharing* a region of *physical memory*?
- *Sharing* is easy: call `mmap()` with `MAP_SHARED` flag
- *Agreeing* is the problem
  Solution: use a *file name* as a meeting point
- Minor problem... one may not want to waste disk space for transient data (not persistent accross system shutdown)
  `MAP_ANONYMOUS` solves this problem... but looses the association between the file and memory region

Implementation in Linux

- Shared memory files are single inodes located in a specific *pseudo-file-system*, mounted under `/dev/shm`
- Must link the program with `-lrt` (real-time library)

System Call: shm_open()

Open and Possibly Create a POSIX Shared Memory File

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

int shm_open(const char *name, int flags, mode_t mode);
```

Description

- Analogous to open(), but for files specialized into "shared memory rendez-vous", and not mapped to persistent storage
- Argument name must begin with a "/" and may not contain any other "/"
- Arguments flags and mode allow for a subset of their values for open()

  flags: O_RDONLY, O_RDWR, O_CREAT, O_TRUNC, O_NONBLOCK, but *not* O_WRONLY, O_APPEND, or any other flag
  Note: FD_CLOEXEC flag is set automatically

  mode: S_IRUSR, S_IWUSR, S_IXUSR, etc.

System Call: `shm_unlink()`

Unlink a POSIX Shared Memory File

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

int shm_unlink(const char *name);
```

Description

- Shared memory files have *kernel* persistence
- Similar to `unlink()`
- `close()` works as usual to close the file descriptor after the memory mapping has been performed
- Neither `close()` nor `unlink()` impact shared memory mapping themselves

About Pointers in Shared Memory

Caveat of Virtual Memory

1. Pointers are variables whose value is a *virtual memory address*

About Pointers in Shared Memory

Caveat of Virtual Memory

1. Pointers are variables whose value is a *virtual memory address*
2. Virtual memory is mapped differently in every process

About Pointers in Shared Memory

Caveat of Virtual Memory

1. Pointers are variables whose value is a *virtual memory address*
2. Virtual memory is mapped differently in every process
3. Consequence
   In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment

About Pointers in Shared Memory

Caveat of Virtual Memory

1. Pointers are variables whose value is a *virtual memory address*
2. Virtual memory is mapped differently in every process
3. Consequence
   In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment
4. Big problem for *linked data structures* and function pointers

About Pointers in Shared Memory

Caveat of Virtual Memory

1. Pointers are variables whose value is a *virtual memory address*
2. Virtual memory is mapped differently in every process
3. Consequence
   In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment
4. Big problem for *linked data structures* and function pointers
5. Mapping to a specified address (the `start` argument of `mmap()`) is a fragile solution

About Pointers in Shared Memory

Caveat of Virtual Memory

1. Pointers are variables whose value is a *virtual memory address*
2. Virtual memory is mapped differently in every process
3. Consequence
   In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment
4. Big problem for *linked data structures* and function pointers
5. Mapping to a specified address (the `start` argument of `mmap()`) is a fragile solution
6. Making pointers relative to the base address of the segment is another solution (cumbersome: requires extra pointer arithmetic)

About Pointers in Shared Memory

Caveat of Virtual Memory

1. Pointers are variables whose value is a *virtual memory address*
2. Virtual memory is mapped differently in every process
3. Consequence
   In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment
4. Big problem for *linked data structures* and function pointers
5. Mapping to a specified address (the `start` argument of `mmap()`) is a fragile solution
6. Making pointers relative to the base address of the segment is another solution (cumbersome: requires extra pointer arithmetic)
7. Note: the problem disappears when forking *after* mapping the shared memory segment

Concurrent Resource Management

Concurrency Issues

- Multiple *non-modifying accesses* to *shared resources* may occur in parallel without conflict
- Problems arise when *accessing a shared resource* to *modify its state*
  - Concurrent file update
  - Concurrent shared memory update
- General problem: enforcing *mutual exclusion*

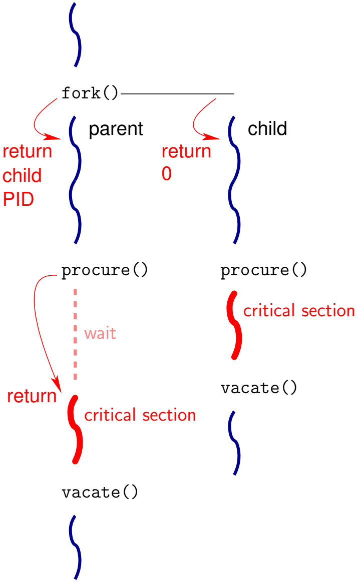
Principles of Concurrent Resource Management

Critical Section

- Program section accessing shared resource(s)
- *Only one process can be in this section at a time*

Mutual Exclusion

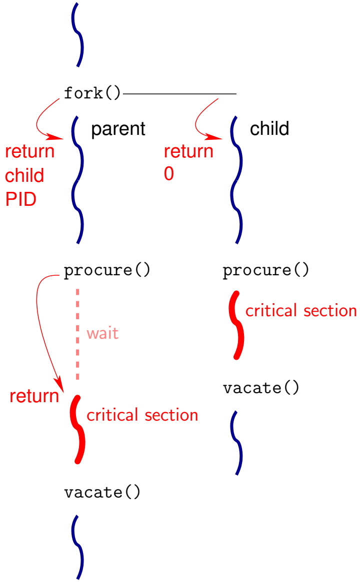
- Make sure at most one process may enter a critical section
- Typical cases
  - Implementing file locks
  - Concurrent accesses to shared memory



Principles of Concurrent Resource Management

Source of Major Headaches

- *Correctness*: prove process alone in critical section
- *Absence of dead-lock*, or detection and lock-breaking
- *Guaranteed progress*: a process enters critical section if it is the only one to attempt to do it
- *Bounded waiting*: a process waiting to enter a critical section will eventually (better sooner than later) be authorized to do so
- *Performance*: reduce overhead and allow parallelism to scale



Mutual Exclusion in Shared Memory

Dekker's Algorithm

```
int try0 = 0, try1 = 0;
int turn = 0;    // Or 1

// Fork processes sharing variables try0, try1, turn
```

```
// Process 0
try0 = 1;
while (try1) {
  if (turn != 0) {
    try0 = 0;
    while (turn != 0) { }
    try0 = 1;
  }
}
// Critical section
// ...
turn = 0;
try0 = 0;
// Non-critical section
```

```
// Process 1
try1 = 1;
while (try0) {
  if (turn != 1) {
    try1 = 0;
  while (turn != 1) { }
    try1 = 1;
  }
}
// Critical section
// ...
turn = 1;
try1 = 0;
// Non-critical section
```

Mutual Exclusion in Shared Memory

Peterson's Algorithm

```
int try0 = 0, try1 = 0;
int turn = 0;     // Or 1

// Fork processes sharing variables try0, try1, turn
```

```
// Process 0
try0 = 1;
turn = 0;
while (try1 && turn == 1) { }
// Critical section
// ...
try0 = 0;
// Non-critical section
```

```
// Process 1
try1 = 1;
turn = 1;
while (try0 && turn == 0) { }
// Critical section
// ...
try1 = 0;
// Non-critical section
```

- Unlike Dekker's algorithm, enforces fair turn alternation
- Simpler and easily extensible to more than two processes

Memory Consistency Issues

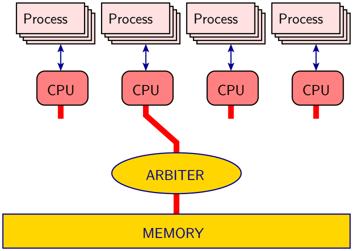
Sequential Consistency

- Semantical chacterization of “when the outcome of a shared memory access is made visible to other processes”
- The preceding algorithms require the strongest (practical) model of memory consistency: *sequential consistency*
- Definition of sequential consistency by Leslie Lamport:
  “*The results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*”

Memory Consistency Issues

Sequential Consistency

- Semantical chacterization of "when the outcome of a shared memory access is made visible to other processes"
- The preceding algorithms require the strongest (practical) model of memory consistency: *sequential consistency*
- Definition of sequential consistency by Leslie Lamport:
  "*The results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"



Memory Consistency Issues

Sequential Consistency

- Semantical chacterization of "when the outcome of a shared memory access is made visible to other processes"
- The preceding algorithms require the strongest (practical) model of memory consistency: *sequential consistency*
- Definition of sequential consistency by Leslie Lamport:
  "*The results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"

Weak Consistency Models

- Pracical hardware, run-time libraries and programming languages enforce *weaker consistency models*
  - Due to compiler optimizations
    Loop-invariant code motion, instruction reordering, etc.
  - Hardware support for out-of-order memory transactions
    Superscalar execution (local), cache coherence (multi-processor)

Memory Consistency Issues

Sequential Consistency

- Semantical chacterization of "when the outcome of a shared memory access is made visible to other processes"
- The preceding algorithms require the strongest (practical) model of memory consistency: *sequential consistency*
- Definition of sequential consistency by Leslie Lamport:
  "*The results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"

Critical Section Algotithms for Weak Memory Consistency

- Other methods avoid (expensive) sequentially consistent reads to shared variables
  E.g., Lamport's "bakery algorithm"

Memory Consistency Issues

Paradoxical Example

```
int f = 1;
int x = 0;

// Fork processes sharing variables f, x

// Process 0
while (f) { }
printf("x = %d\n", x);
```

```
// Process 1
x = 1;
f = 0;
```

Analysis

- What is the value of `x` printed by `Process 0`?
  (assuming no other process may access the shared variables)

Memory Consistency Issues

Paradoxical Example

```
int f = 1;
int x = 0;

// Fork processes sharing variables f, x

// Process 0                               // Process 1
while (f) { }                              x = 1;
printf("x = %d\n", x);                     f = 0;
```

Analysis

- What is the value of x printed by Process 0?
  (assuming no other process may access the shared variables)
  - **1** with sequential consistency

Memory Consistency Issues

Paradoxical Example

```
int f = 1;
int x = 0;

// Fork processes sharing variables f, x

// Process 0                                    // Process 1
while (f) { }                                   x = 1;
printf("x = %d\n", x);                          f = 0;
```

Analysis

- What is the value of `x` printed by `Process 0`?
  (assuming no other process may access the shared variables)
  - **1** with sequential consistency
  - May be **0** with weaker models

Memory Consistency Issues

Paradoxical Example

```
int f = 1;
int x = 0;

// Fork processes sharing variables f, x

// Process 0                              // Process 1
while (f) { }                             x = 1;
printf("x = %d\n", x);                    f = 0;
```

Analysis

- What is the value of `x` printed by `Process 0`?
  (assuming no other process may access the shared variables)
  - **1** with sequential consistency
  - May be **0** with weaker models
  - May even be **42**!

Solution: Hardware Support

Serializing Memory Accesses

- Memory *fences* (for the hardware and compiler)
  - Multiprocessor
    $\longrightarrow$ Commit all pending memory and cache coherence transactions
  - Uniprocessor (cheaper and weaker)
    $\longrightarrow$ Commit all local memory accesses
  - Can be limited to read or write accesses
  - Forbids cross-fence code motion by the compiler
- ISO C `volatile` attribute (for the compiler)
  - `volatile int x`
    Informs the compiler that asynchronous modifications of `x` may occur
  - No compile-time reordering of accesses to volatile variables
  - Never consider accesses to volatile variables as dead code

- Combining fences and volatile variables fixes the problems of Dekker's and Peterson's algorithms
- Modern programming languages tend to merge both forms into more abstract constructs (e.g., Java 5)

Solution: Hardware Support

Atomic Operations

- Fences are expensive (especially on parallel architectures)
- Finer grained *atomic operations* permit higher performance
  - *Exchange*:
    Atomically exchange the values of a register and a memory location
  - *Test-and-Set*:
    Atomically tests a memory location, set it to **1**, and return whether the old value was null or not
    Can be implemented with atomic exchange

```
int test_and_set(int *lock_pointer) {
  int old;
  old = atomic_exchange(lock_pointer, 1);
  return old != 0;
}
```

  - Many others, often implementable with atomic exchange, but may involve direct processor support for higher performance

Semaphore

Unified Structure and Primitives for Mutual Exclusion

- Initialize the semaphore with `v` instances of the resource to manage

  ```
  void init(semaphore s, int v) {
    s.value = v;
  }
  ```

- Acquire a resource (entering a critical section)

  ```
  void procure(semaphore s) {
    wait until (s.value > 0);
    s.value--;      // Must be atomic with the previous test
  }
  ```

  Also called `down()` or `wait()`

- Release a resource (leaving a critical section)

  ```
  void vacate(semaphore s) {
    s.value++;      // Must be atomic
  }
  ```

  Also called `up()` or `post()`

Implementation of a Simple Lock

```c
void procure(volatile int *lock_pointer) {
  while (test_and_set(lock_pointer) == 1);
}
void vacate(volatile int *lock_pointer) {
  *lock_pointer = 0 // Release lock
}

int lock_variable = 1; // Or any non-negative number
{
  procure(&lock_variable);
  // Critical section
  // ...
  vacate(&lock_variable);
}
```

Semaphores (Multiple Resource Instances)

- Use atomic decrement and increment instructions
- Or use simple lock to protect counter incrementations/decrementations

Heterogeneous Read-Write Mutual Exclusion

Read-Write Semaphores

- Allowing *multiple readers* and a *single writer*

```
void init(rw_semaphore l) {
  l.value = 0;   // Number of readers (resp. writers)
                 // if positive (resp. negative)
}

void procure_read(rw_semaphore l) {
  wait until (l.value >= 0);
  l.value++;     // Must be atomic with the previous test
}
void vacate_read(rw_semaphore l) {
  l.value--;     // Must be atomic
}

void procure_write(rw_semaphore l) {
  wait until (l.value == 0);
  l.value = -1;  // Must be atomic with the previous test
}
void vacate_write(rw_semaphore l) {
  l.value = 0;
}
```

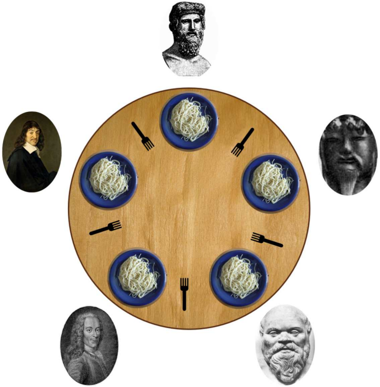
Mutual Exclusion and Deadlocks

Dining Philosophers Problem

- Due to Edsger Dijkstra and Tony Hoare
  - Eating requires two chopsticks (more realistic than forks...)
  - A philosopher may only use the closest left and right chopsticks

  Multiple processes acquiring multiple resources

- Typical case of deadlock: all philosophers pick their left chopstick, *then* attempt to pick their right one



Mutual Exclusion and Deadlocks

Avoiding Deadlocks

- Avoid symmetric acquire/release patterns
- Use higher-level mutual exclusion mechanisms
  - Monitors
  - Atomic transactions
- Debugging non-reproducible dead-locks is difficult

Breaking Deadlocks

1. Timeout
2. Analyze the situation
3. Attempt to reacquire different resources or in a different order

Beyond Deadlocks

- Livelocks (often occurs when attempting to break a deadlock)
- Aim for fair scheduling: bounded waiting time
- Stronger form of fairness: avoid priority inversion in process scheduling

IPC: Semaphores

POSIX Semaphores

- Primitives: `sem_wait()` (`procure()`) and `sem_post()` (`vacate()`)
  - `sem_wait()` blocks until the value of the semaphore is greater than **0**, then decrements it and returns
  - `sem_post()` increments the value of the semaphore and returns
- They can be named (associated to a file) or not
- `$ man 7 sem_overview`

Implementation in Linux

- Semaphore files are single inodes located in a specific *pseudo-file-system*, mounted under `/dev/shm`
- Must link the program with `-lrt` (real-time library)

System Call: sem_open()

Open and Possibly Create a POSIX Semaphore

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int flags);
sem_t *sem_open(const char *name, int flags,
                mode_t mode, unsigned int value);
```

Description

- Arguments flags and mode allow for a subset of their values for open()

  flags: O_CREAT, O_EXCL, but *not* O_RDONLY, O_RDWR, O_WRONLY,
  O_APPEND, O_TRUNC, O_NONBLOCK, or any other flag
  Note: FD_CLOEXEC flag is set automatically

  mode: S_IRUSR, S_IWUSR, S_IXUSR, etc.

- value is used to initialize the semaphore
  Defaults to **1** if not specified

- Returns the address of the semaphore on success
- Returns SEM_FAILED on error (i.e., (sem_t*)-1)

System Call: sem_wait()

Lock a POSIX Semaphore

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

Description

- Blocks until the value of the semaphore is greater than $\mathbf{0}$, then decrements it and returns
- Returns $\mathbf{0}$ on success, $-\mathbf{1}$ on error

System Call: sem_post()

Unlock a POSIX Semaphore

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Description

- Increments the value of the semaphore pointed to by sem
- Returns **0** on success, **−1** on error

System Call: sem_close()

Close a POSIX Semaphore Structure

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Description

- Similar to close() for semaphore pointers
- Undefined behavior if closing a semaphore that other processes are currently blocked on

System Call: sem_unlink()

Unlink a POSIX Semaphore File

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Description

- Semaphores files have *kernel* persistence
- Similar to unlink()

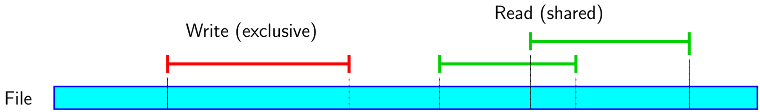
Other System Calls

- sem_init and sem_destroy: create unnamed semaphores and destroy them (equivalent to combined sem_close() and sem_unlink())
- sem_getvalue: get the current value of a semaphore
- sem_trywait and sem_timedwait: non-blocking and timed versions of sem_wait

Alternative: I/O Synchronization With Locks

Purpose

- Serialize processes accessing the same region(s) in a file
- When at least *one process is writing*
- Two kinds of locks: *read* (a.k.a. *shared* and *write* (a.k.a. *exclusive*)
- Two independent APIs supported by Linux
  - POSIX with `fcntl()`
  - BSD with `flock()`



I/O System Call: `fcntl()`

Manipulate a File Descriptor

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

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, struct flock *lock);
```

Main Commands

`F_DUPFD`: implements `dup()`

`F_GETLK`/`F_SETLK`/`F_SETLKW`: acquire, test or release *file region* (a.k.a. *record*) lock, as described by third argument `lock`

I/O System Call: `fcntl()`

Manipulate a File Descriptor

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

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, struct flock *lock);
```

Return Value

- On success, `fcntl()` returns a (non-negative) value which depends on the command, e.g.,

  `F_DUPFD`: the new file descriptor

  `F_GETLK`/`F_SETLK`/`F_SETLKW`: **0**
- Returns $-\mathbf{1}$ on error

I/O System Call: `fcntl()`

Manipulate a File Descriptor

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

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, struct flock *lock);
```

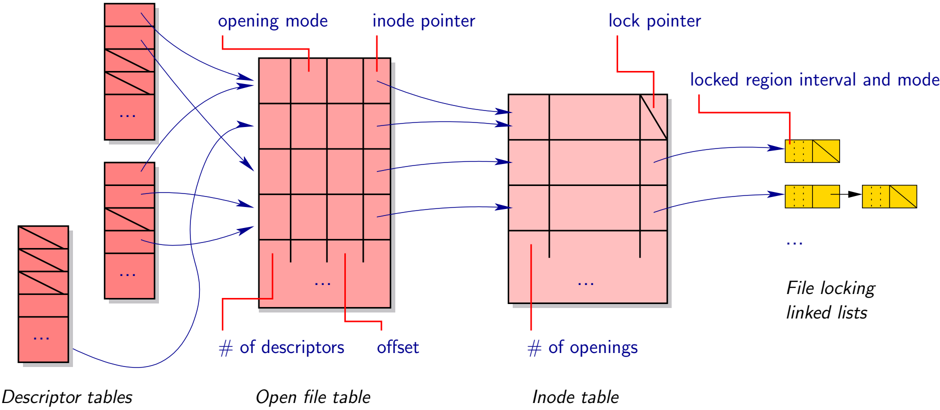
About File Locks

- `fcntl()`-style locks are POSIX locks; they are *not* inherited upon `fork`
- BSD locks, managed with the `flock()` system call, are inherited by `fork()`
- Both kinds are *advisory*, preserved across `execve()`, fragile to `close()` (releases locks), removed upon termination, and supported by Linux
- `$ man 2 fcntl` and `$ man 2 flock`
- Linux supports SVr3 mandatory `fcntl()`-style locks,
  - Depending on file system mounting options (`-o (no)mand`)
  - Disabled by default: very deadlock-prone (especially on NFS)
  - Linux prefers *leases* (adds signaling and timeout)

More About File Locks

Implementation Issues

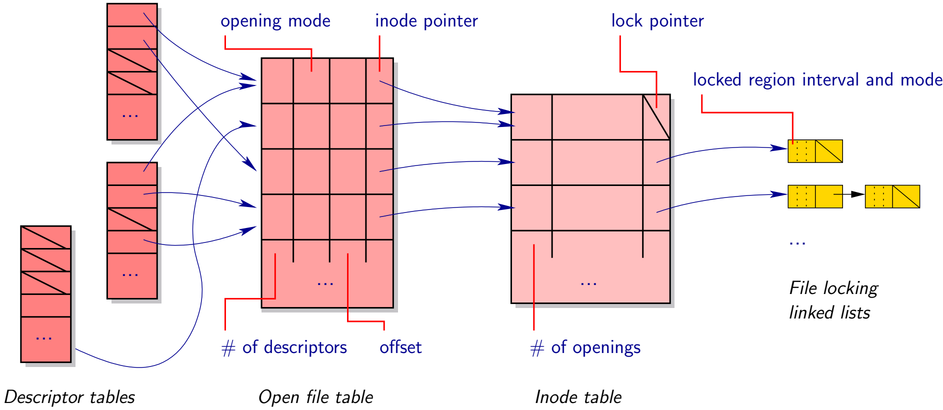
- Locks are associated with *open file entries*:
  → They are lost when closing all descriptors related to a file



More About File Locks

Implementation Issues

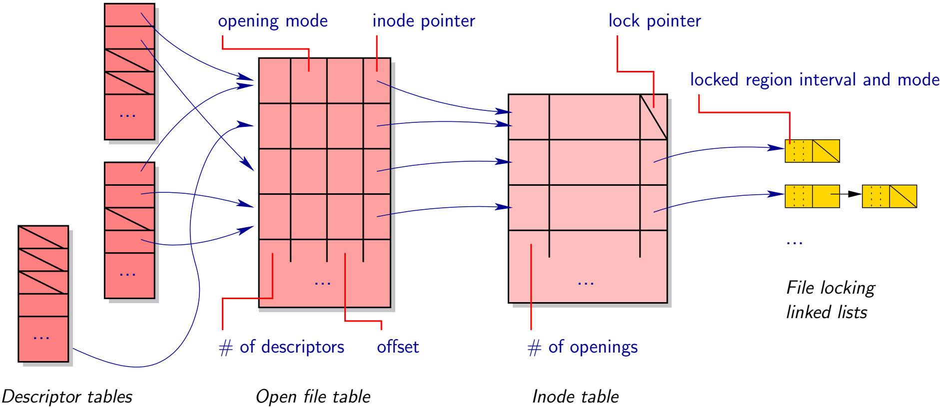
- They are incompatible with C library I/O (advisory locking and buffering issues)



More About File Locks

Consequences for the System Programmer

- File locks may be of some use for *cooperating* processes only
- Then, why not use *semaphores*?



System V IPC

Old IPC Interface

- Shared motivation with POSIX IPC
  - Shared memory segments, message queues and semaphore sets
  - Well-defined semantics, widely used, but widely criticized API
  - `$ man 7 svipc`
- But poorly integrated into the file file system
  - Uses (hash) *keys* computed from unrelated files
    `$ man 3 ftok`
  - Conflicting and non-standard naming
  - Ad-hoc access modes and ownership rules
- Eventually deprecated by POSIX IPC in 2001