Inter-Processes Communication (IPC)

Modern Operating Systems, by Andrew Tanenbaum — Chap 2

Operating Systems: Three Easy Pieces (a.k.a. the OSTEP book) — Chap 4&5

Primary IPC styles

- Pipes
  - Uni-directional passing of data (if used cleanly)
  - e.g. ls –l | more
- Signals
  - Event notification from one process to another
- Shared Memory
  - More than one process can read and write to the same memory
  - Requires synchronization between the sharing processes!

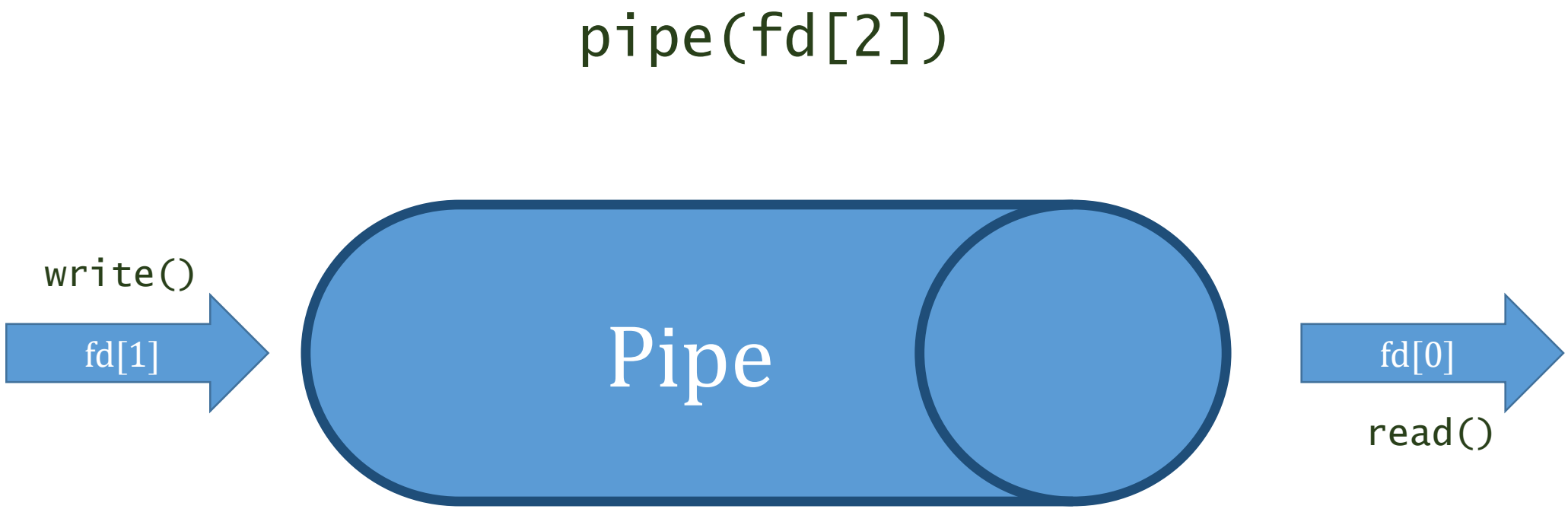
Other forms of IPC

- Parent/Child IPC
  - Command line arguments
  - Process return codes wait(…), waitpid(…) and exit(…)
- Reading/Writing common files
  - Servers use "/run/xyz.pid" file to determine other active servers
- Semaphores
  - Locking and event signaling mechanisms between processes
- Sockets
  - Bi-directional
  - Not just across the network, but also between processes

Pipes

Pipe Abstraction

- Write to one end, read from another



UNIX : Files vs. Streams

- The original C standard library supports "Files Descriptors"
  - A file-descriptor is an integer index into a "File Descriptor table"
  - Functions like open, close, read, write, pipe use file-descriptor indexes
- Streams are a more sophisticated I/O model built on top of file descriptors
  - A stream is (usually) a pointer to a FILE structure
  - The FILE structure contains a file-descriptor index, plus more
  - Functions like fopen, fclose, fread, fwrite, popen, and pclose use steams
- Both File Descriptors and Steams can be "attached" to hard disk files, devices (such as keyboards and monitors), network, etc.

File-Descriptor Table

- Each process has a file-descriptor table
  - One entry for each open file descriptor

Index	File Descriptor
0	stdin
1	stdout
2	stderr
?	fd[0]
?+1	fd[1]

File Descriptors and Fork

- When a fork occurs, the entire address space, *including the file descriptor table*, is cloned!

Index	File Descriptor
0	stdin
1	stdout
2	stderr
?	fd[0]
?+1	fd[1]

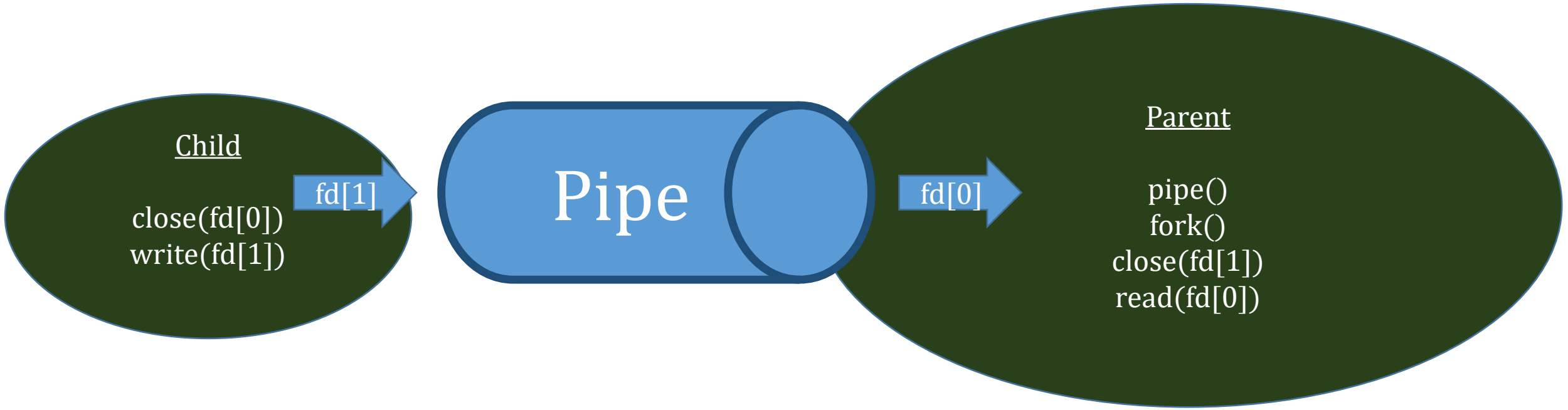
Index	File Descriptor
0	stdin
1	stdout
2	stderr
?	fd[0]
?+1	fd[1]

Duplicating file descriptors

- `int dup(oldfd)`
  - duplicates file descriptor with index x, returns the index of the duplicate
  - Makes an exact copy of the file descriptor, including position in the "file"
    - Well something of a cross between copy and reference

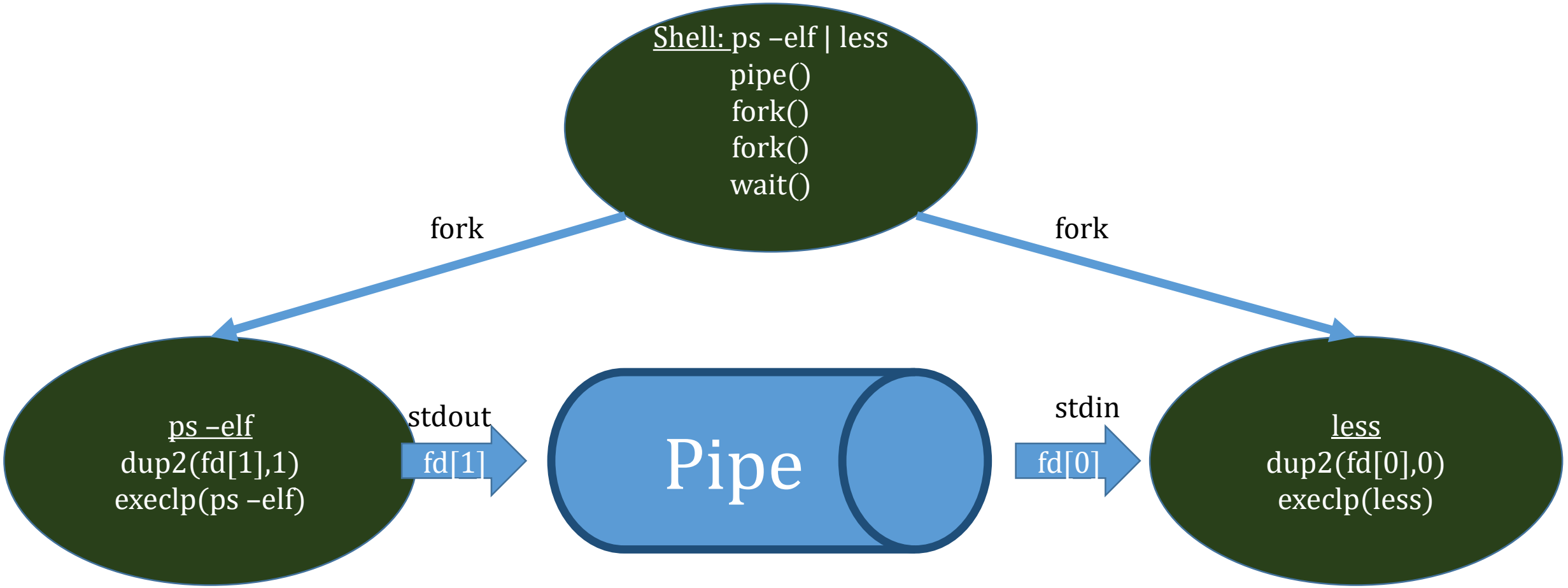
- `int dup2(oldfd,newfd)`
  - if (newfd==oldfd) returns newfd
  - if newfd is open, closes newfd
  - duplicates oldfd to newfd
  - `dup2(fds[0], STDIN_FILENO);  /* make stdin same as fds[0] */`
  - `dup2(fds[1], STDOUT_FILENO);  /* make stdout same as fds[1] */`

Parent/Child Process Pipes



See: Examples/ipc/pipe1.c

Shell Pipes


Shell: ps –elf | less
pipe()
fork()
fork()
wait()
fork
fork
ps –elf
dup2(fd[1],1)
execlp(ps –elf)
stdout
fd[1]
Pipe
stdin
fd[0]
less
dup2(fd[0],0)
execlp(less)

Chains of Filters – Recursive approach

- Split chain into head "|" tail
- Create a pipe
- Fork a child
- Redirect stdout/stdin using dup2 in parent and child
- exec head in child
- if tail is not empty, parent invokes function recursively on tail
- if tail is empty, parent waits on all children

Pipes : byte-stream abstraction

- You can read from or write to a pipe at arbitrary byte boundaries
  - E.g. write 10 bytes, 10 bytes, 10 bytes
  - read 5 bytes 15 bytes 15 bytes 5 bytes

- Message abstraction imposes message boundaries
  - E.g. network packets

Reading/Writing with pipes

- `read(fds[0],buf,6);`
  - May not read 6 bytes!
  - Why?

- Some reasons:
  - read() could reach end of input stream (EOF)
  - Other side of pipe may abruptly close the connection (broken pipe)
  - read() could return on a signal

- You **MUST** use error handling with any system call (including I/O)

Example of Error Handling

You must

- Check the return from every system call
- Then either handle errors OR
- continue processing

- Convenient to write wrapper functions

```
/* Write "n" bytes to a  descriptor. */
ssize_t writen(int fd, const void *vptr, size_t n) {
    size_t nleft; size_t nwritten; const char *ptr;
    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft))<=0){
        if (errno == EINTR) nwritten = 0; /* call write() again*/
        else return(-1); /* error */
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return(n);
}
```

Signals

What is a signal?

- A notification to a process that an (asynchronous) event has occurred
  - May originate in this process, another process, or from the OS who manages external devices
- There is no data associated with a signal
  - A signal either occurs or does not occur
- There is a pre-defined set of signals supported by UNIX
  - Send signals to a process with the "kill" command
  - List all signal types with the command "kill –l"
- Some interesting signals: SIGCHILD, SIGKILL, SIGSTOP

Handling Signals

- When a signal is sent to a process, the OS stops that process from executing, saves the execution state, and transfers control to a signal handler routine
- The operating system signal handler routines typically print an error message and exit the process (e.g. SIGSEGV)
- SIGSTOP halts execution, but leaves the process idle
  - SIGCONT signal causes execution to continue
- If the signal handler routine returns, the process state is restored, and the process continues to execute (e.g. SIGCHLD and SIGURG)

Overriding Default Signal Handler

- The C library function sigaction(…) allows us to specify a different action for a specific signal
  - *except* SIGKILL and SIGSTOP cannot be overiden!
  - Note: sigaction uses C function pointers.

- For example, long running programs may use SIGUSR1 to report current status
- For example, program may want to handle SIGSEGV like a C version of try/catch

See: Examples/ipc/signals_ex.c

More on SIGCHLD

- When child process terminates or stops, SIGCHLD is sent to parent
- By default, parent ignores SIGCHLD
- Override default by running sigaction() for SIGCHLD before parent calls fork()
  - If act.sa_handler is SIG_IGN, SIGCHLD will be ignored by parent
  - If act.sa_flags is SA_NOCLDSTOP, SIGCHLD will not be generated by child
  - If act.sa_flags is SA_NODLDWAIT, children will not become zombies when they terminate

How to avoid waitpid blocking

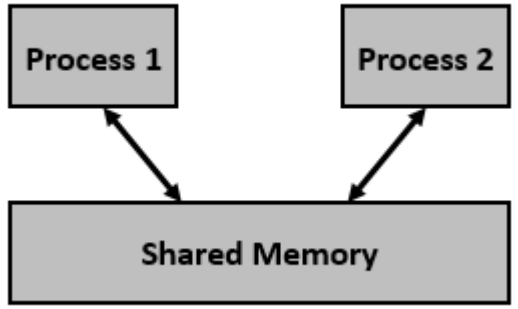
- Parent installs signal handler for SIGCHLD
- Signal handler invokes wait(…) or waitpid(…)

```
/* SIGCHLD handler */
void int_handler(int sig) {
  pid_t pid;
  int stat;
  pid = wait(&stat); // Never blocks!
  printf("In Parent: Child %d  terminated\n", pid);
  printf("Parent continues...\n");
}
```

See: Examples/ipc/sigchld.c

More Info

- Check "man sigaction"
- What happens when a signal occurs while a signal handler is running?
- What happens when a signal is delivered in the middle of a system call? (Different OS's have different behaviors)
- Google: "Unix Signals"… tons of useful links



Shared Memory

Shared Memory Library functions: shmget, shmat, shmdt, shmctl

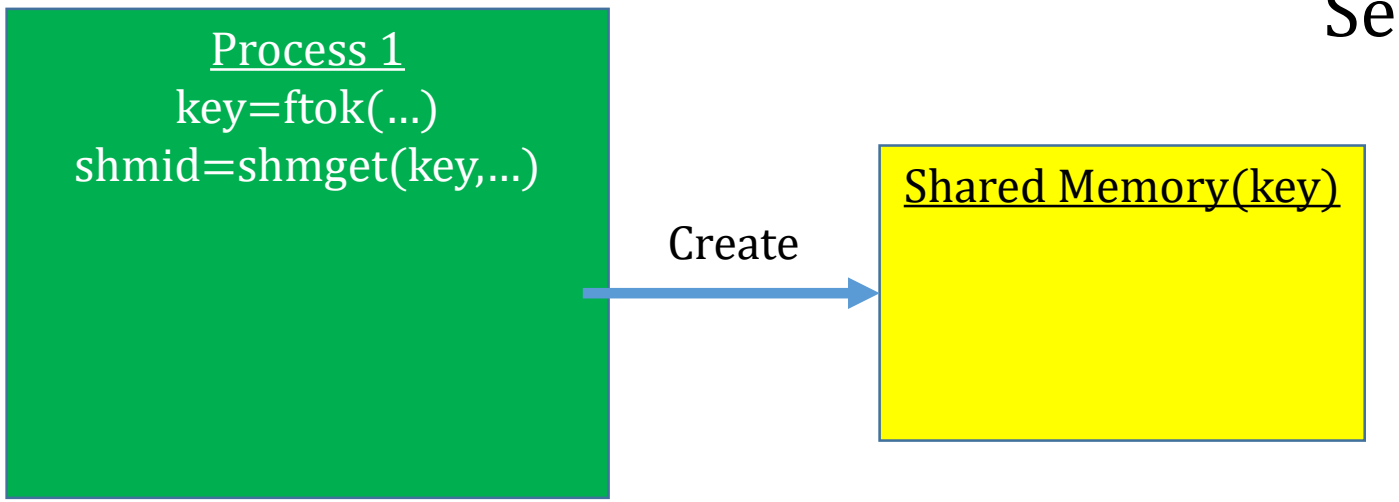
Shared Memory Concepts

- Use ftok(…) to get a unique shared memory key
  - ftok parameter : File name (controls access),
  - ftok parameter: projid (enables multiple keys with a single file)

- Key is then used to reserve or connect to a block of memory with shmget(…), which returns a schmid
- Memory can be attached to your process address space with shmat(…) using schmid
- Memory can be deleted with shmctl(…)

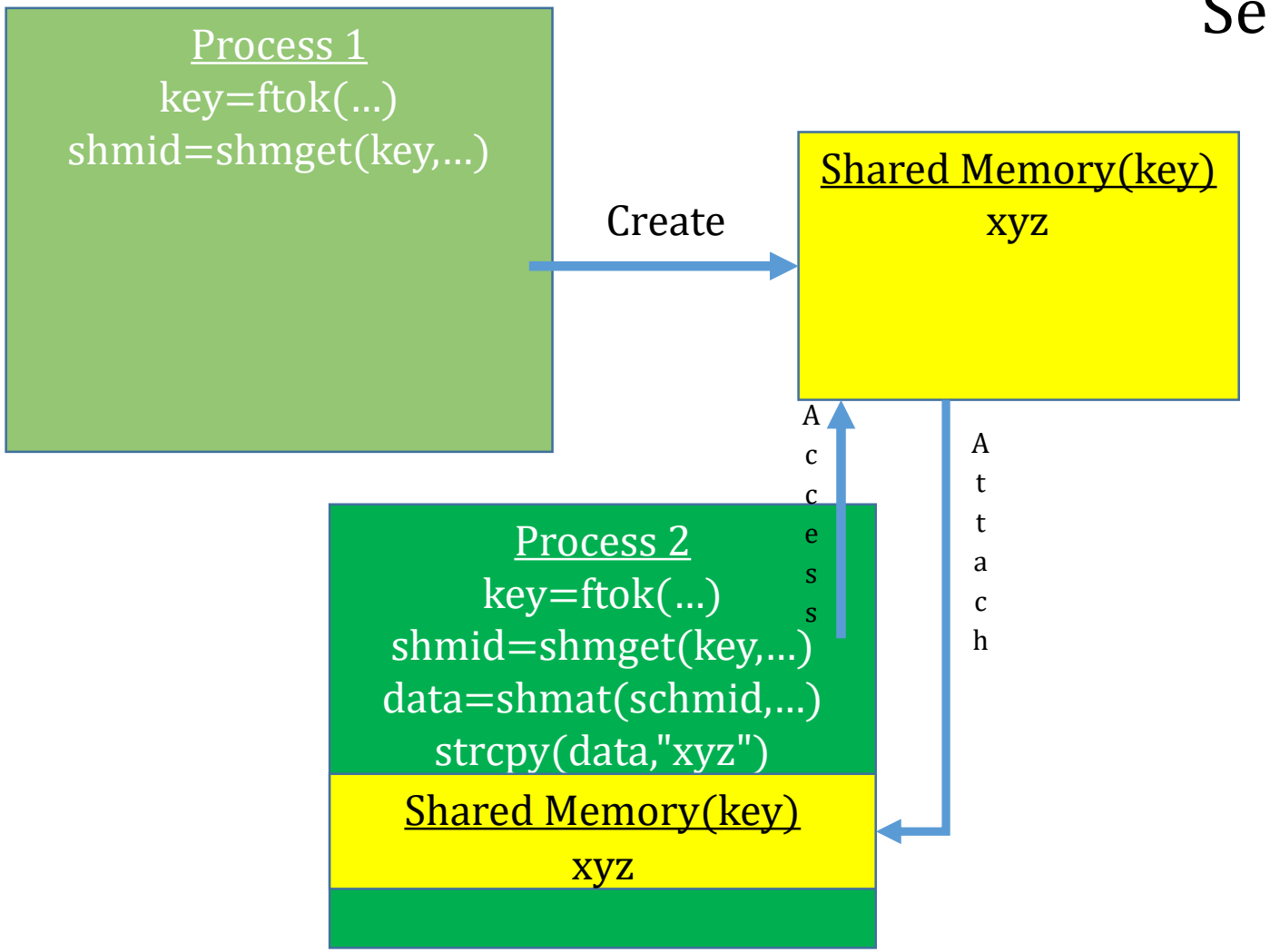
Creating shared memory

See: Examples/ipc/sh_create.c



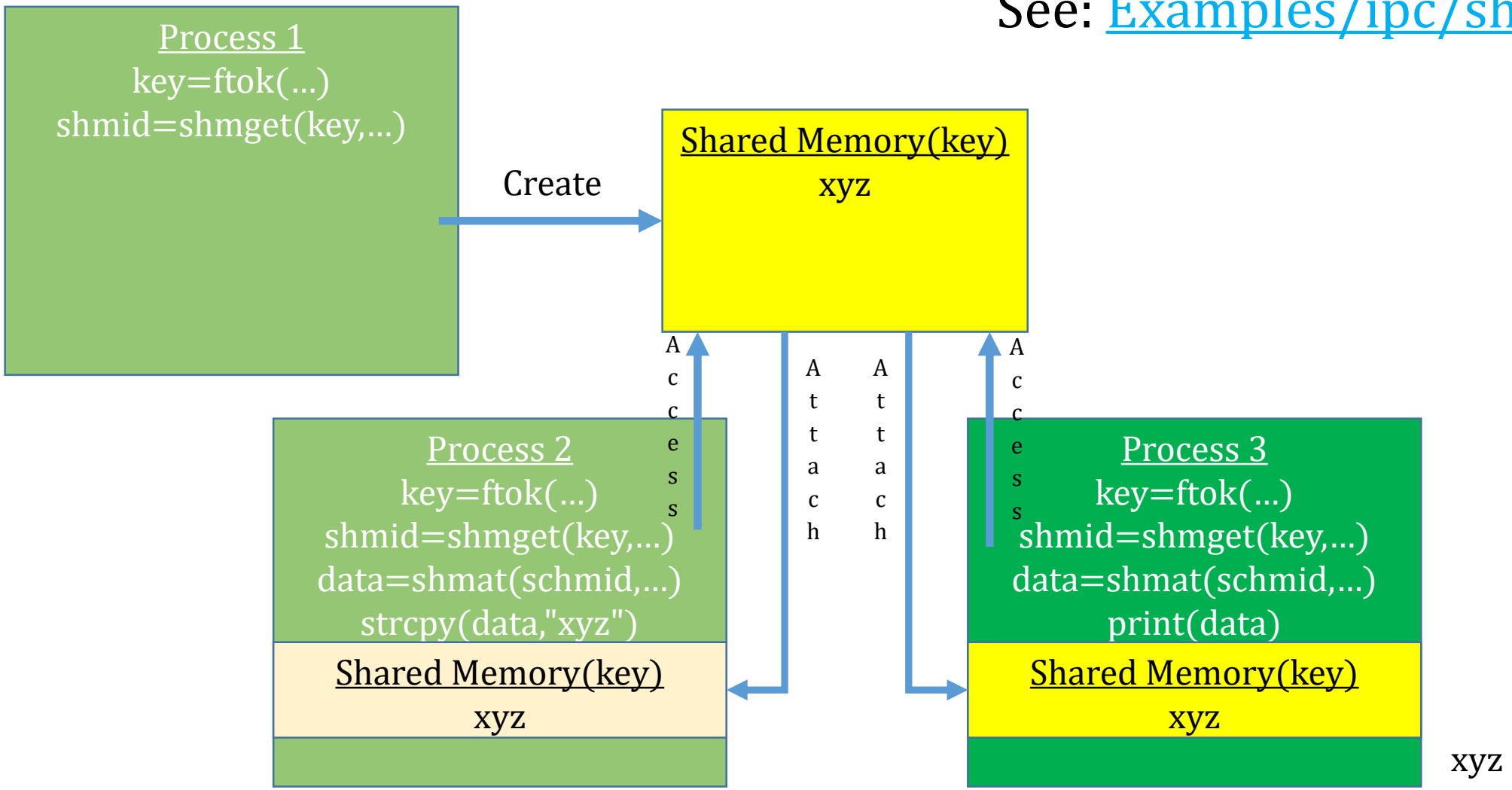
Accessing shared memory

See: Examples/ipc/sh_access.c



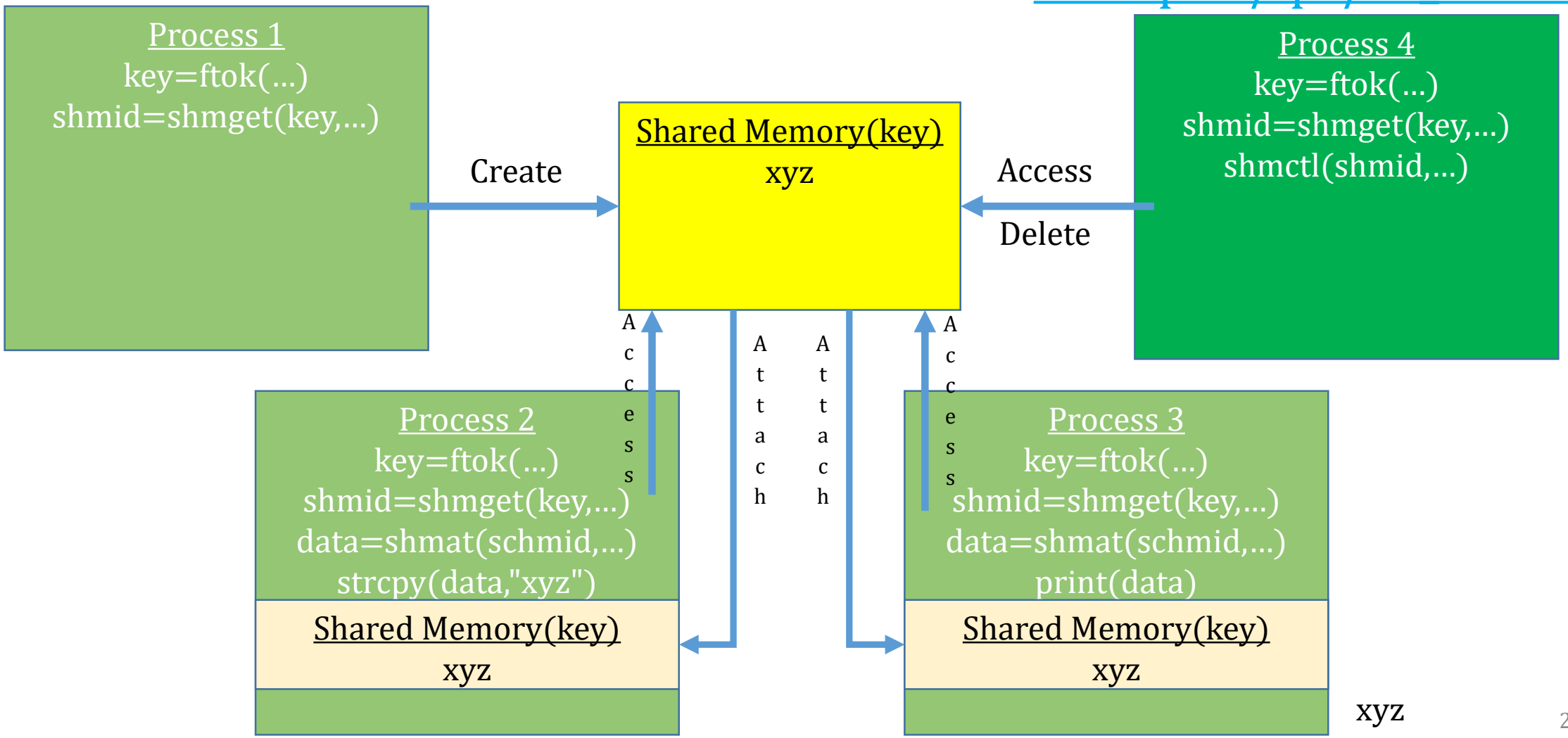
Accessing Shared Memory

See: Examples/ipc/sh_access.c



Deleting shared memory

See: Examples/ipc/sh_delete.c



Shared Memory Notes

- shmdet (shared memory detach) not shown above – see code.

- IPC commands:
  - ipcs – lists all IPC objects owned by the user
  - ipcrm – removes specific IPC object

- References
  - Unix man pages
  - "Advanced Programming in Unix Environment" by Richard Stevens http://www.kohala.com/start/apue.html

Shared Memory Warning

- C assumes it owns memory
  - If your program hasn't changed a variable, it hasn't changed
- With shared memory, this assumption is incorrect
  - A different process may change the value of the memory
- At a minimum, mark variables in shared memory as "volatile"

- If processes are truly concurrent, need more control

```
if (shared!=0) result = 276/shared; // divide by zero error!
```

- Semaphores (yet to come)

File Descriptors Addendum

File Descriptor

- An integer index into a process specific "File Descriptor Table" kept in the kernel
- Therefore, file descriptors are non-negative integers
  - Negative integers are used to indicate an error
- File descriptors are normally created by an "open()" call, but may also be created by other calls (such as dup or dup2 or pipe)
- An entry in the file descriptor table contains:
  - The position in the file (offset from the beginning of the file)
  - flags: Binary bits to manage things like append, async, cloexec, create,…
  - A pointer to a system-wide "File Table" (if the file is open)
- Use "fctl()" to modify a file descriptor table entry
- See /proc/*pid*/fdinfo for the list of file descriptors associated with *pid*

System Wide File Table

- Contains ONE entry for every "file" open for ALL processes!
  - Multiple File Descriptor table entries all point to ONE file table entry
    - e.g. after fork, both parent file descriptor tables and child file descriptor tables point to the same File table entry
    - e.g. after dup or dup2, both the old and new fd point to the SAME File table entry!
- Keeps track of the mode the file was opened (read or write or RW)
- Keeps track of the NUMBER of file descriptor entries which are "open" and which point to this File table entry
- When you do a close, the number is decremented, and the pointer from the file descriptor to the file table is removed
- When that number reaches zero, the file is ACTUALLY closed!

Implications of this Design

- Suppose I have a pipe created to pass data from one child to another child
- When the first child closes the input side of the pipe, I want the second child to see an end-of-file
- If ALL fd's associated with the write end of the pipe are not closed, the read end of the pipe will not see an end-of-file!
  - Parent must close input side of pipe
  - Both children must close input side of pipe
  - If dup or dup2 is called for input side, both old and new fd's must be closed!