Processes

Chapter 3

With the execution environment, we are ready to create a process

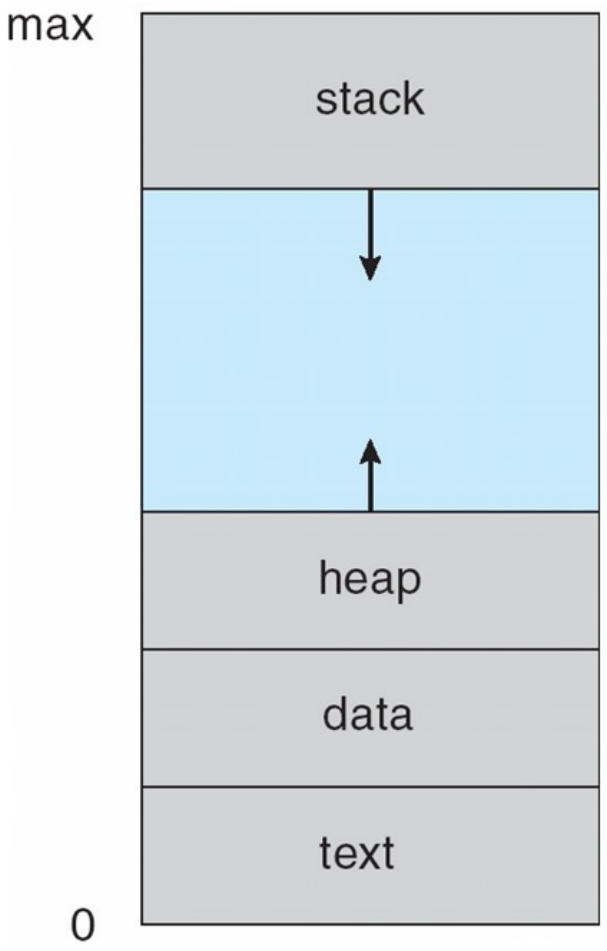
- A *Process* is an abstraction for a running program, including:
  - Ability to execute instructions
  - Protected region of memory
  - Ability to have and use resources
  - Usually, some associated protection domain
    - e.g., the privileges of the user who created it
- Program is a bunch of instructions (and maybe some static data)

OS APIs for Process Management

- Supports creating, destroying, querying process state, etc.
- Interleaves execution of multiple processes on CPUs
- Provides mechanisms for Inter-Process Communication (IPC)
- Provides mechanisms to allocate resources to processes

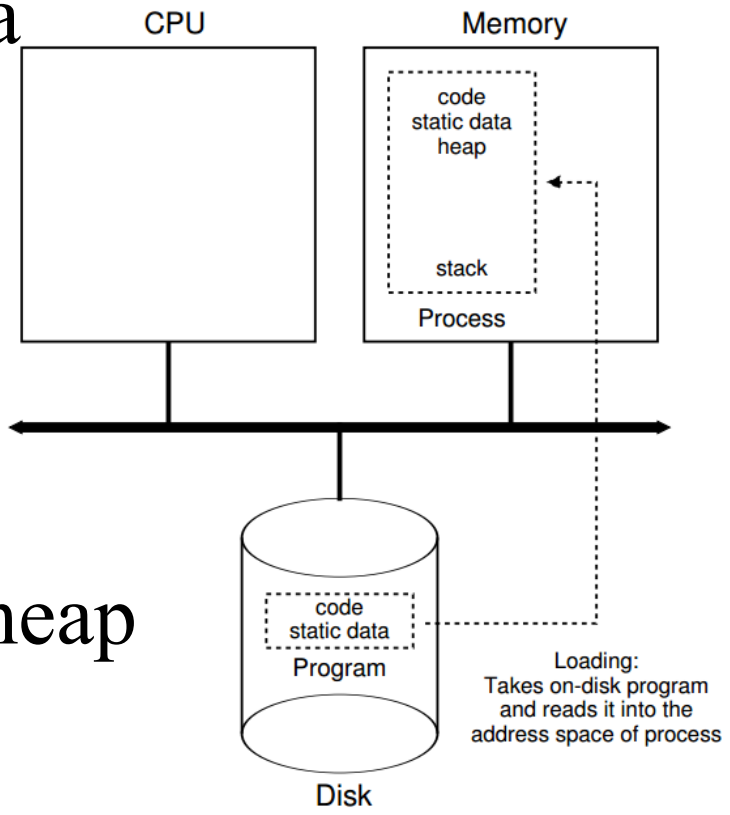
Process Memory Layout (simplified)

- Text section: code and other global, read-only data
- Data: writable global variables
- Stack: local variables, return addresses, etc.
- Heap: dynamically allocated memory (malloc(), new)
- Room for stack / heap to grow



Process Creation

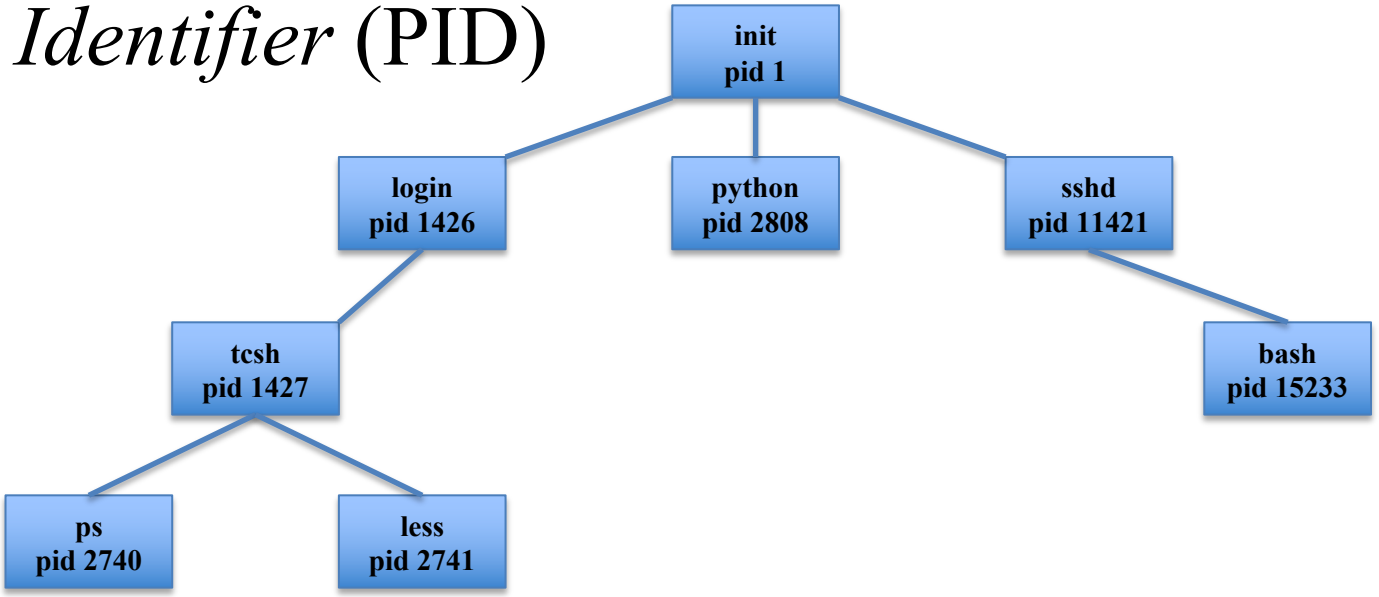
- Loading: code and static data
  - Eagerly vs lazily



- Allocate memory for stack, heap
- Initialize file descriptors
- Jump to the main and give the CPU to the process

Where do Processes Come From?

- After boot, existing processes create new ones
  - *Parent* process
  - *Child* processes
  - Tree of processes (rooted at init in Unix)
- Each with a unique *Process Identifier* (PID)



Do Children Look Like Their Parents?

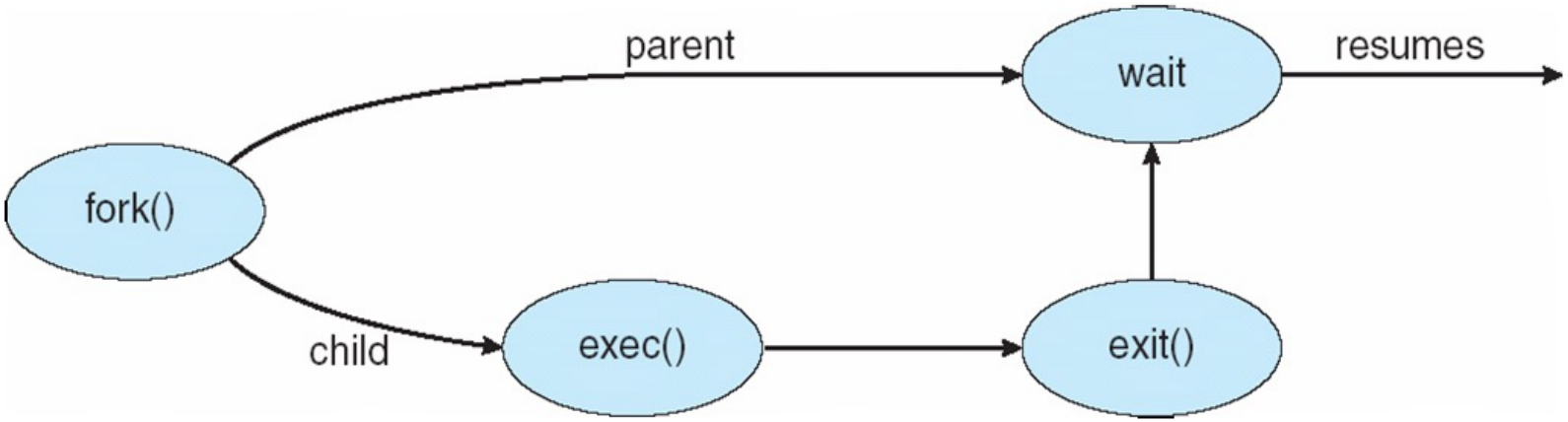
- Inheriting resources (alternatives)
  - Parent and child share all resources
  - Child inherits some of the parent's resources
    - e.g., POSIX, child inherits open files
  - Parent and child share no resources
- Memory (alternatives)
  - Child is a duplicate of parent
    - e.g., POSIX fork() **then** exec()
  - Child has a new program loaded into its address space
    - e.g., Windows CreateProcess()

Do Children Look Like Their Parents?

- Execution (alternatives)
  - Parent and child run concurrently
  - Or parent waits for child to terminate

Typically, you get this

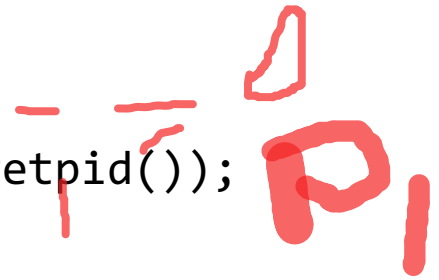
But you can make it behave like this



In-class exercise

Assume the following C code is compiled to create the executable program, pointless, which is then run on a UNIX system. How many processes will be part of the pointless program? Assuming fork() does not return an error.

```
int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc;
    int i = 0;
    for (; i < 3; i++)
      fork();

    printf("hello, I am (pid:%d)\n", (int) getpid());
    return 0;
}
```

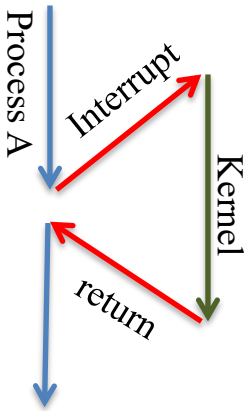


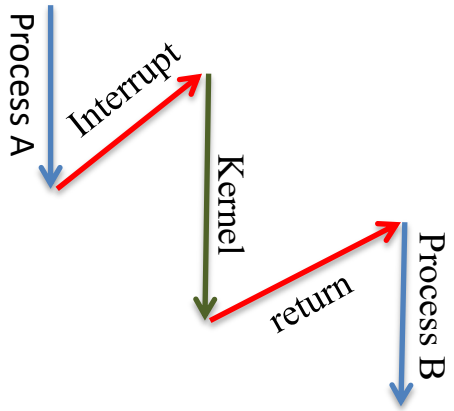
Making Processes Share the CPU

- Processes need a chance to execute
- Remember
  - Once one user process starts executing,
  - … the timer interrupt must fire for the kernel to get a chance to run again
- When the timer interrupt occurs, the CPU:
  - Saves CPU state (at least the program counter)
  - Switches to kernel mode
  - Jumps to an interrupt handler

Making Processes Share the CPU

- Often, the kernel may
  - Return to the running process
- Sometimes, the kernel will
  - Switch to a different process
  - i.e., change where the interrupt return goes
  - This is called a *context switch*

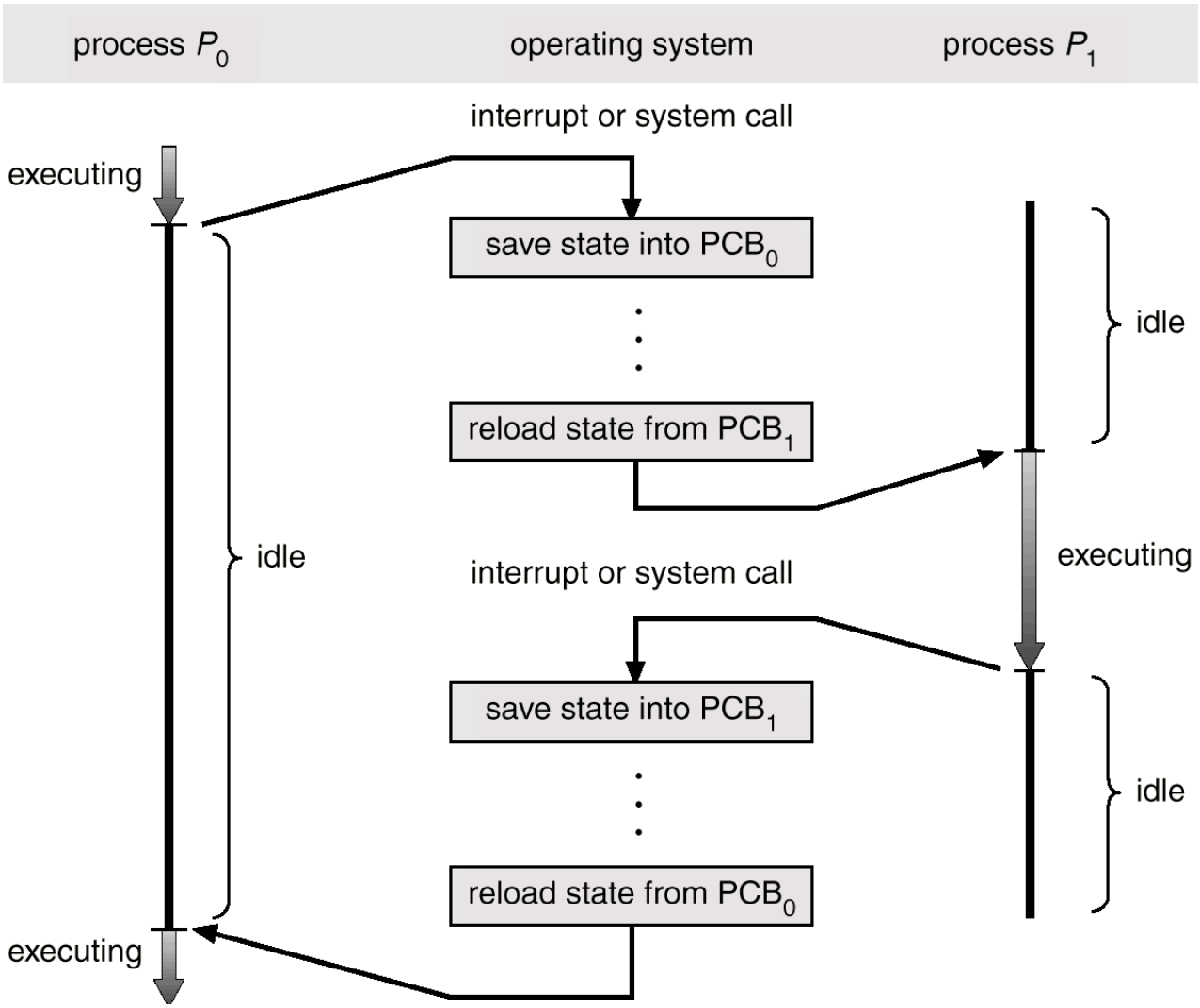




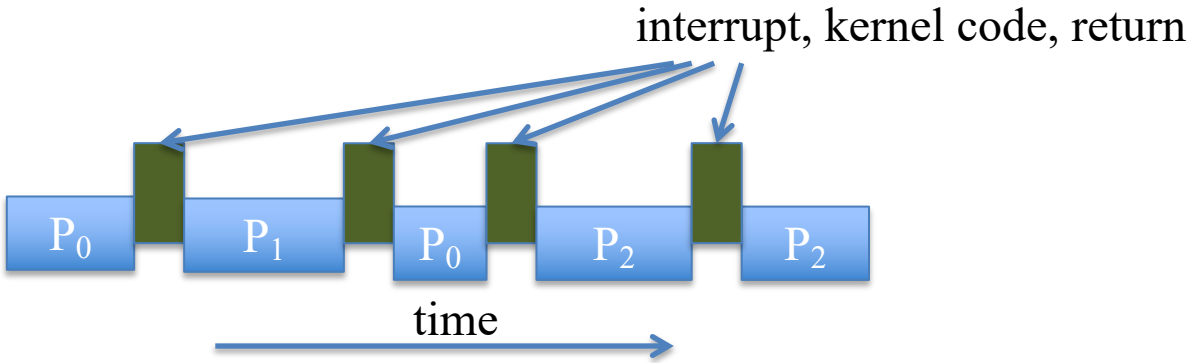
Performing a Context Switch

- Switching from Process P_0 to Process P_1
  - Interrupt occurs (PC pushed to stack, switch to kernel mode, jump to interrupt handler)
  - Decide to run Process P_1 instead (this is a scheduling decision)
  - Save a copy of PC and other CPU registers for P_0 (somewhere)
  - Save memory bounds for P_0 (e.g., base and limit)
  - Restore memory bounds for P_1
  - Restore CPU registers for P_1
  - Copy P_1 PC into stack
  - Return from interrupt

Time-Division Multiplexing the CPU



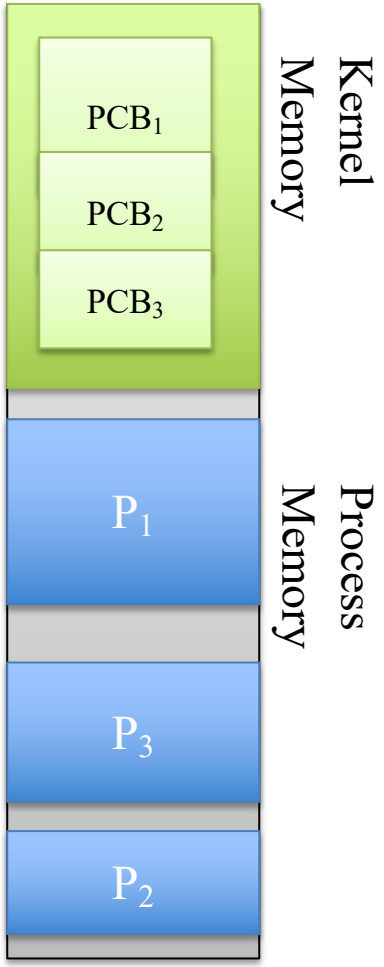
Minding the Gaps



- Kernel gets to run in the gaps, between execution of user processes
- That's OK, running user processes is what the system is for
- Context switch time is all overhead (so we want it to be fast)

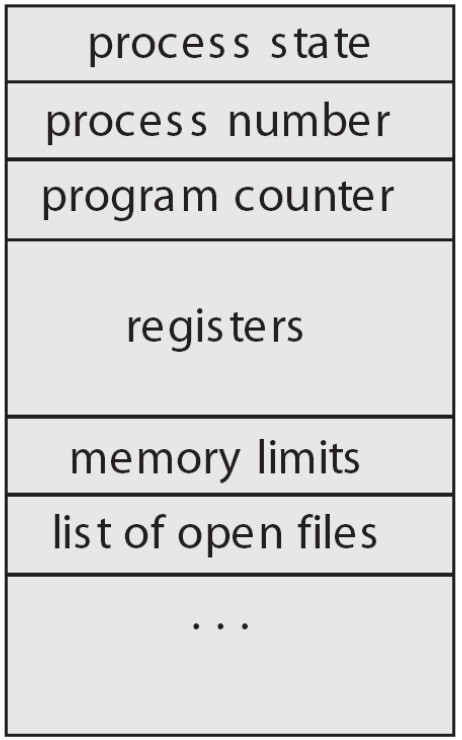
The Process Control Block

- For each process, the kernel needs a small record of information
  - This is where the OS save CPU registers and other information when it does a context switch



Inside The PCB

- Process State
- Unique *Process Identifier (PID)*, also maybe parent PID
- Copies of PC and other CPU registers
- Memory bounds
- Accounting information
- Resources in use (e.g., open files)
- Pointers for linking PCBs into various lists



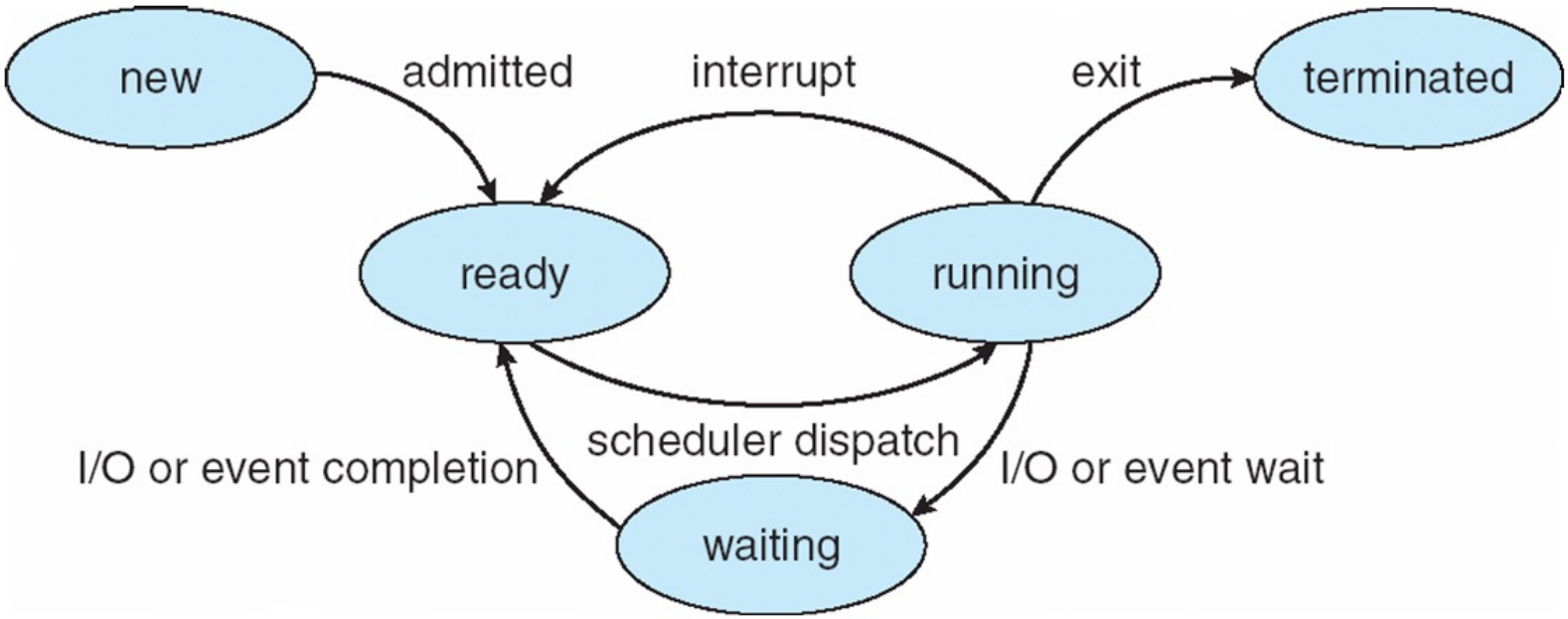
PCB from Linux Kernel (Old, Simplified)

```
struct task_struct {
  volatile long       state;/* -1 unrunnable, 0 runnable, >0 stopped*/
  long                priority;
  int                 errno;
  struct task_struct  *next_task, *prev_task;/* Linked list of tasks*/

  int                 pid;
  unsigned short      uid,euid,suid,fsuid;

  long                utime, stime, cutime, cstime, start_time;
  struct rlimit       rlim[RLIM_NLIMITS];

  struct tty_struct   *tty;           /* NULL if no tty */

  struct files_struct *files;         /* open file information */
  struct mm_struct    *mm;            /* memory management info */
  ...;                                /* Lots of stuff omitted. */
};
```

Process State

- Each process has a *process state*
- State indicates what the OS can/should do with each process
  - *new*: being created and can't be run yet
  - *running*: process is executing instructions on a CPU
  - *waiting*: process has requested I/O (or something else) and can't run until it's complete
  - *ready*: process is runnable, but isn't on a CPU right now
  - *terminated*: process has finished execution, still has a process ID, but isn't runnable

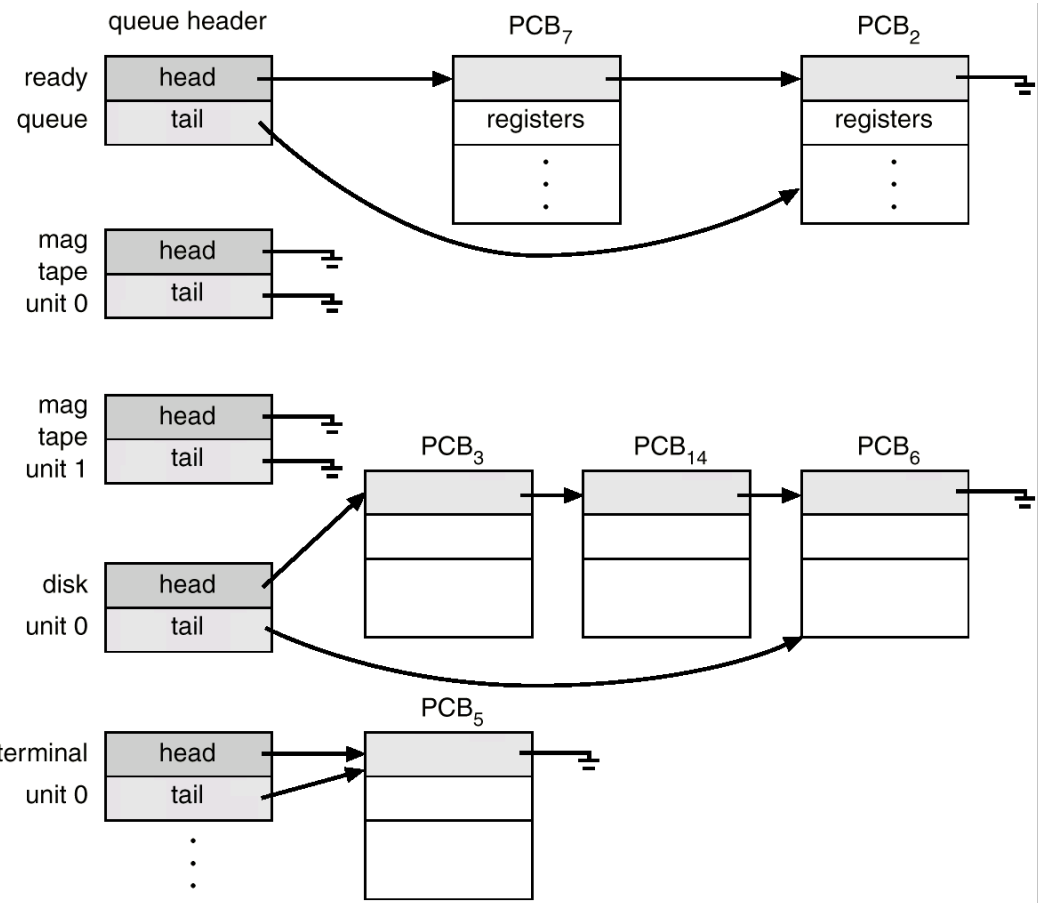
Process State Transitions



- So, how does the kernel find
  - a next runnable process after one completes running?
  - a next process to use the disk after a disk request is completed?

Keeping PCBs Organized

- How are PCBs organized?
  - *Process Table* a list of all PCBs (maybe ordered by PID)
  - *Scheduling Queues* Lists of PCBs for processes waiting on service



Scheduling Queues

- Ready queue
  - Processes waiting to run on the CPU
- Device queues
  - Processes waiting for disk, network or other devices
- OS moves PCBs among these queues
  - When? In the gaps between processes. Whenever an interrupt or trap occurs.
  - Why? To keep resources busy and quickly find a process that's waiting on a particular resource.

Schedulers

- *Short-term scheduler* (CPU scheduler) –
  - Decide which process to execute next
  - Must make quick decisions
- Disk schedule, same idea for a hard disk

Types of Processes

- *CPU – I/O Burst Cycle*
  - A typical process alternates between two states
    - Wants to execute instructions
    - Then, it wants to wait (for I/O or something else)
- *CPU-bound processes*
  - Spend more time doing computations
  - Longer CPU bursts, infrequent or short I/O bursts
- *I/O-bound processes*
  - Spend more time doing I/O than computation
  - Shorter CPU bursts, longer I/O bursts

Process Termination

- A system call to ask for termination
  - exit(), well, really _exit()
  - Called automatically when main() returns
  - Exit status, delivered to the parent via wait()
  - Process resources reclaimed by OS
- When a process terminates, can its children continue?
  - Generally yes.
  - Otherwise, we say we have cascading termination.
- OS may terminate a process (why?)
- Processes may try to terminate each other
  - e.g., task assigned to child is no longer needed
  - POSIX uses signals() to deliver termination requests (and things, it's a general mechanism for delivering notifications)
  - Signals are a mechanism for *upcalls*

POSIX Signal Example

```
static int gotSignal = 0;

void alarmHandler( int sig ) {
  gotSignal = 1;
}

int main() {
  struct sigaction act;

  act.sa_handler = alarmHandler;
  sigemptyset( &( act.sa_mask ) );
  act.sa_flags = 0;
  sigaction( SIGALRM, &act, 0 );

  alarm( 10 );

  char buffer[ 1024 ];
  int len = read( STDIN_FILENO, buffer, sizeof( buffer ) );
  if ( len > 0 )
    printf( "You Said: %.*s", buffer );

  if ( gotSignal )
    printf( "Too Slow\n" );

  return 0;
}
```

Tell the OS to call alarmHandler() when SIGALRM is delivered.

Set an alarm for 10 seconds.

Quick, type a message.

Did we get a signal before you finished?

Inter-Process Communication

- IPC Mechanisms
  - OS provides an environment where processes can be independent
  - But some processes may want to cooperate, say for:
    - Handling multiple clients/requests concurrently
    - Computational speedup with multiple cores
    - Modularity (e.g., shell commands)
    - Isolating less trusted code

Inter-Process Communication

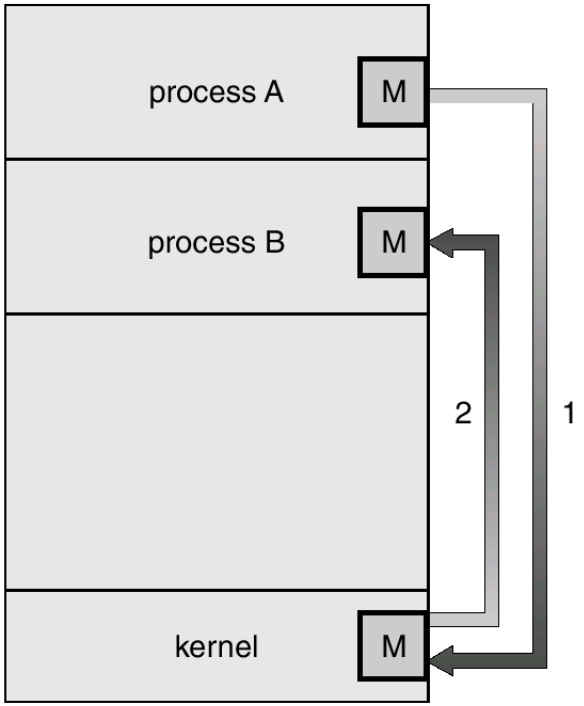
- Two main mechanisms for IPC
  - Message Passing
  - Shared Memory
- Naming: how do we define the sender and recipient?
  - Direct communication, name them in the send/recv
    - send(P0, message);
    - recv(P1, message);
    - Examples, POSIX signals, MPI
  - Indirect communication, send via a named communication channel (a mailbox or a port)
    - send(A, message);
    - recv(A, message);
    - Examples: POSIX pipes, POSIX named pipes, message queues

IPC Design Parameters

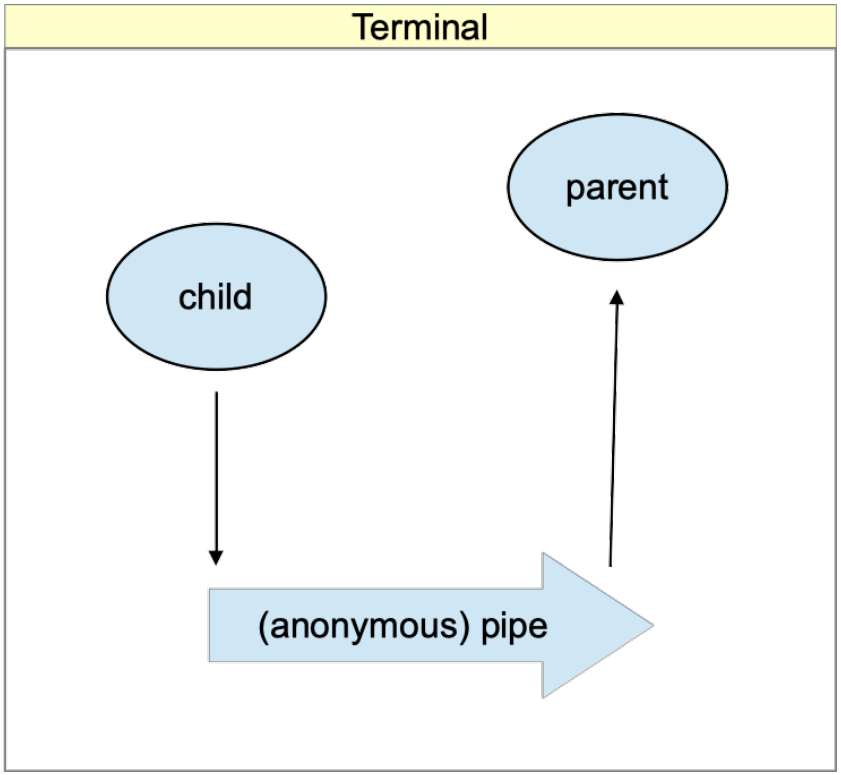
- How are links established?
- Can more than two processes use a link?
- Is the link unidirectional or bi-directional?
- How many links can there be between a pair of processes?
- Is there any buffering?
- Is the size of a message fixed or variable?
- Are message boundaries respected?

Message Passing

- *Message*: a sequence of bytes to exchange
- System calls to:
  - To move message between process' memories
  - Send a message (e.g., send() or just write())
  - Receive a message (e.g., recv() or just read())

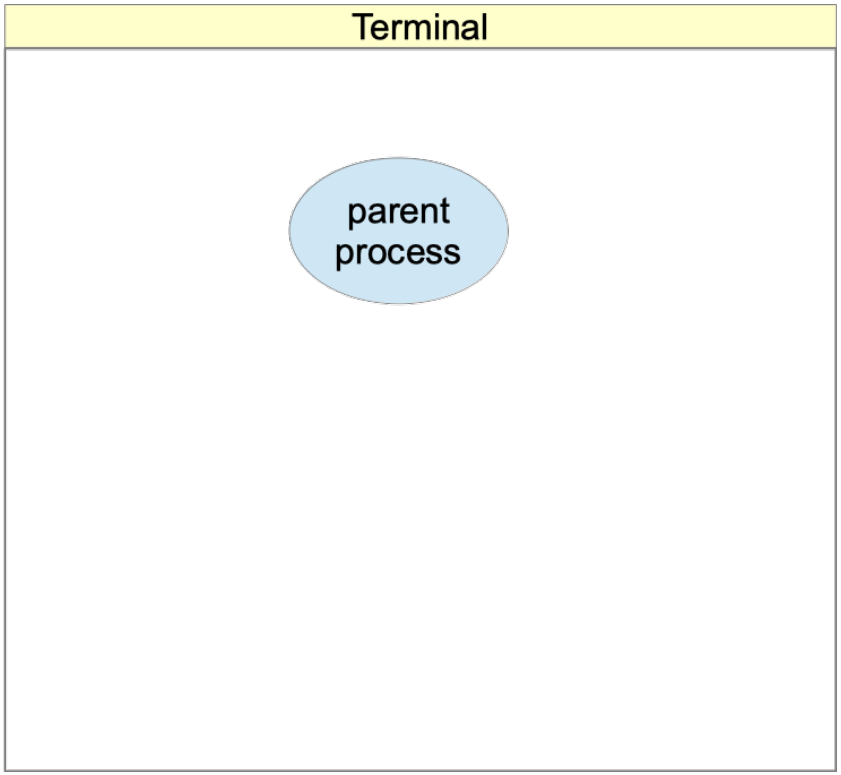


Example: POSIX Anonymous Pipes

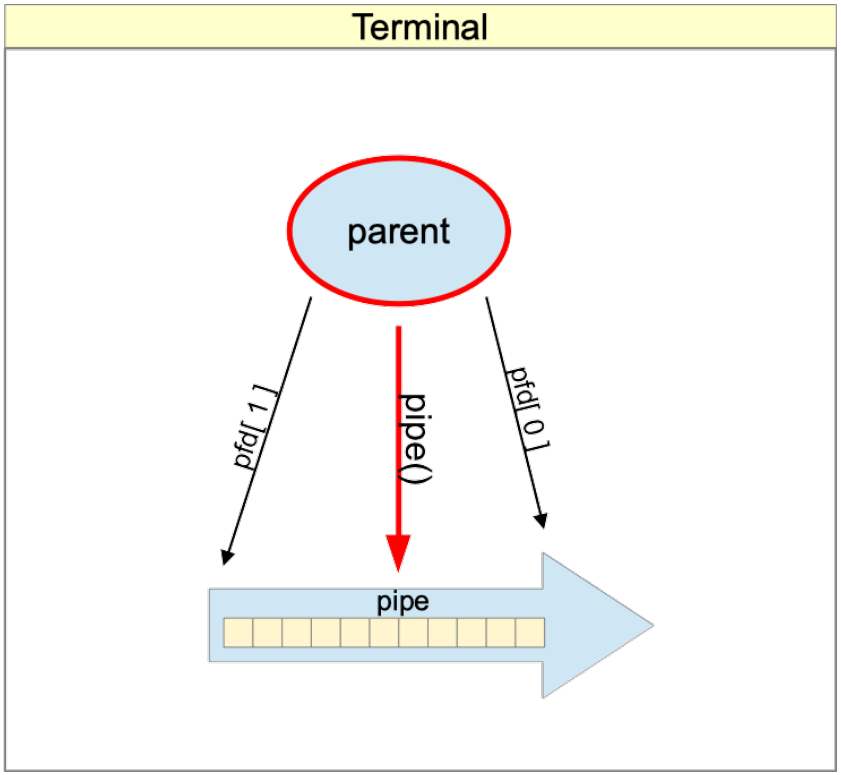


Source file: PipeIPC.c

Parent Process Running in a Terminal

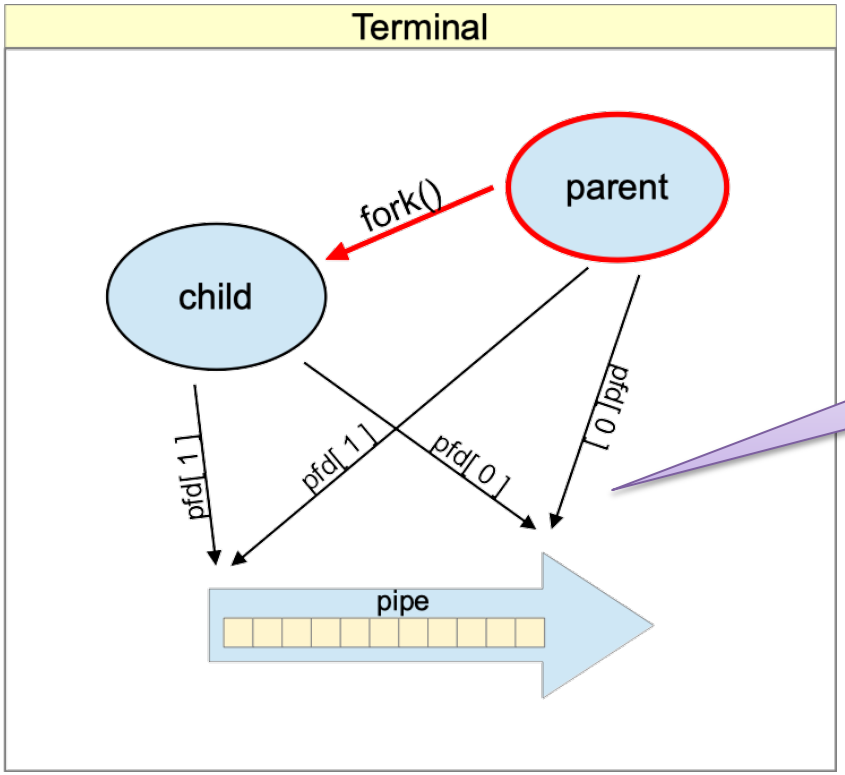


Parent: Makes a Pipe




```
int pfd[ 2 ];
if ( pipe( pfd ) != 0 )
  fail( "Can't create pipe" );
```

Parent: Makes a Child



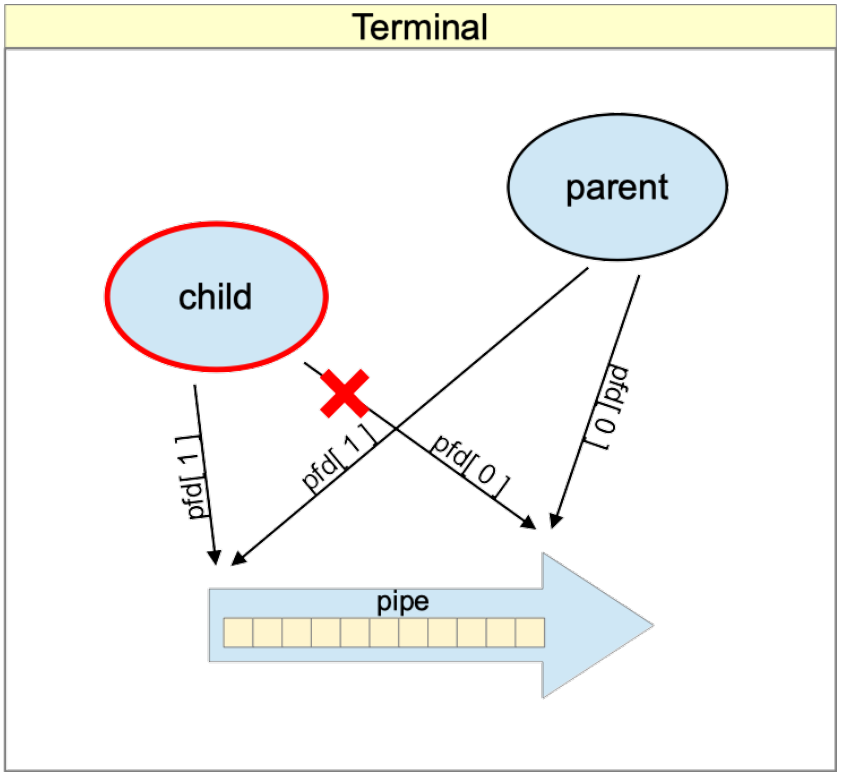
Child inherits the parent's file descriptors.

```
pid_t id = fork();
if ( id == 0 ) {
   ...
} else {
   ...
}
```

child

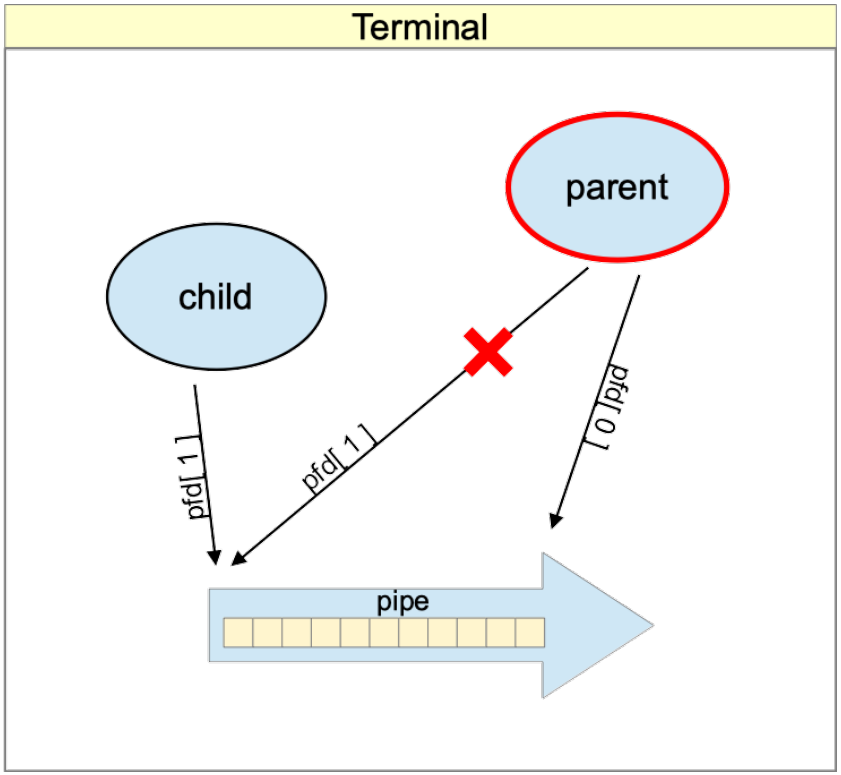
parent

Child: Doesn’t need to Read Pipe




```
if ( id == 0 ) {
  close( pfd[ 0 ] );
} else {
  ...
}
```

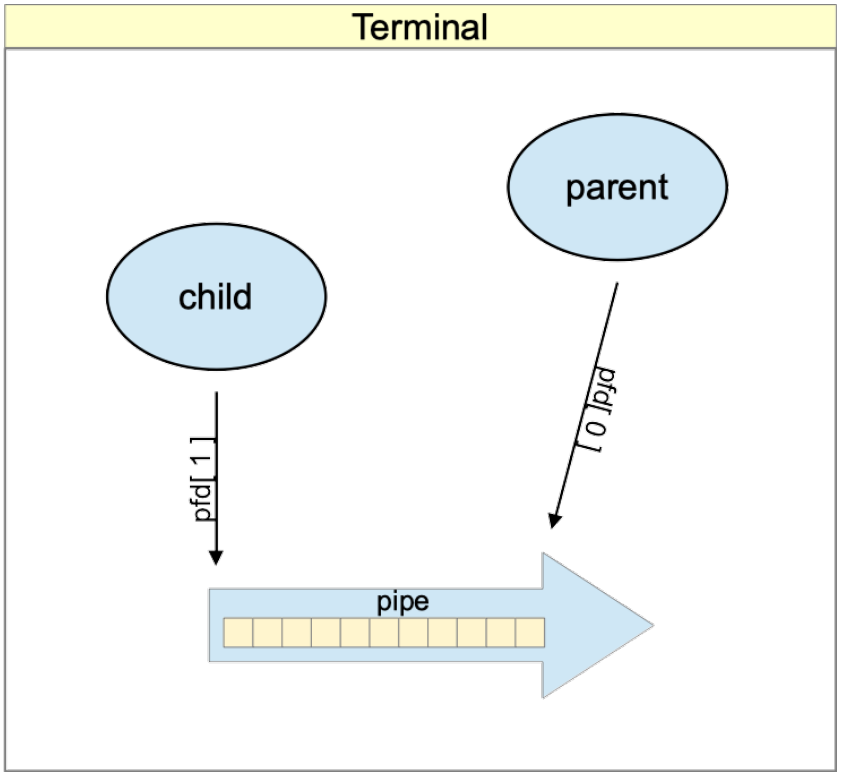
Parent: Doesn't need to Write Pipe



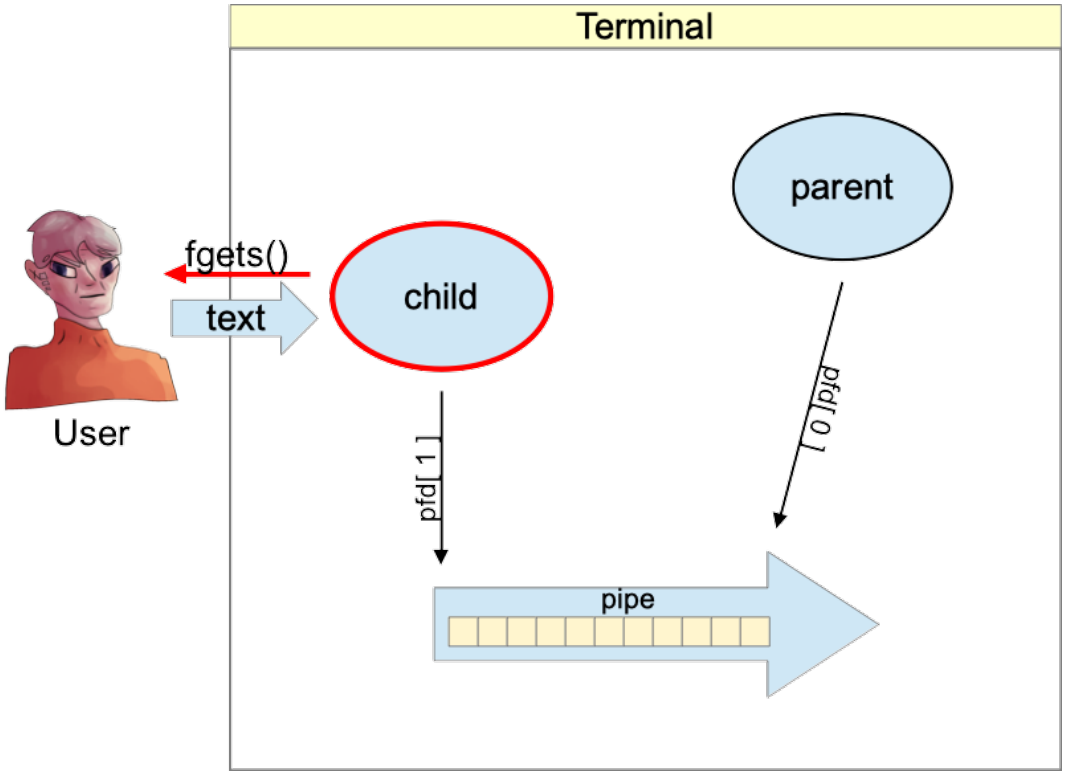
Important, if you expect to reach EOF on the pipe.

```
if ( id == 0 ) {
   ...
} else {
   close( pfd[ 1 ] );
}
```

Parent and Child: Ready to Talk

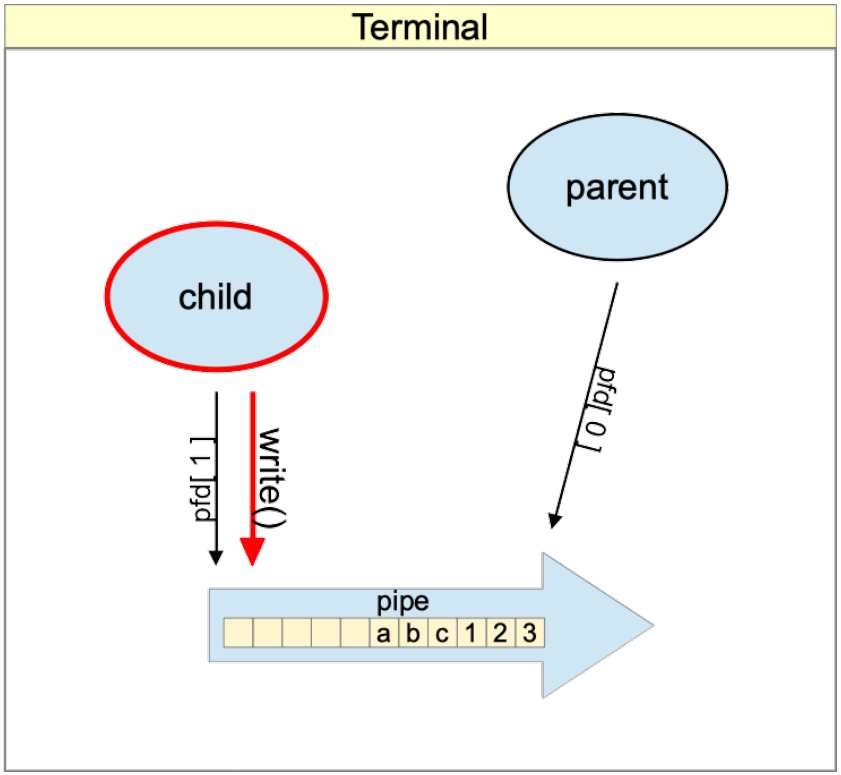


Child: Read a Line from the User




```
if ( id == 0 ) {
   ...
   char buffer[ 1024 ];
   fgets( buffer, sizeof( buffer ), stdin );
} else {
   ...
}
```

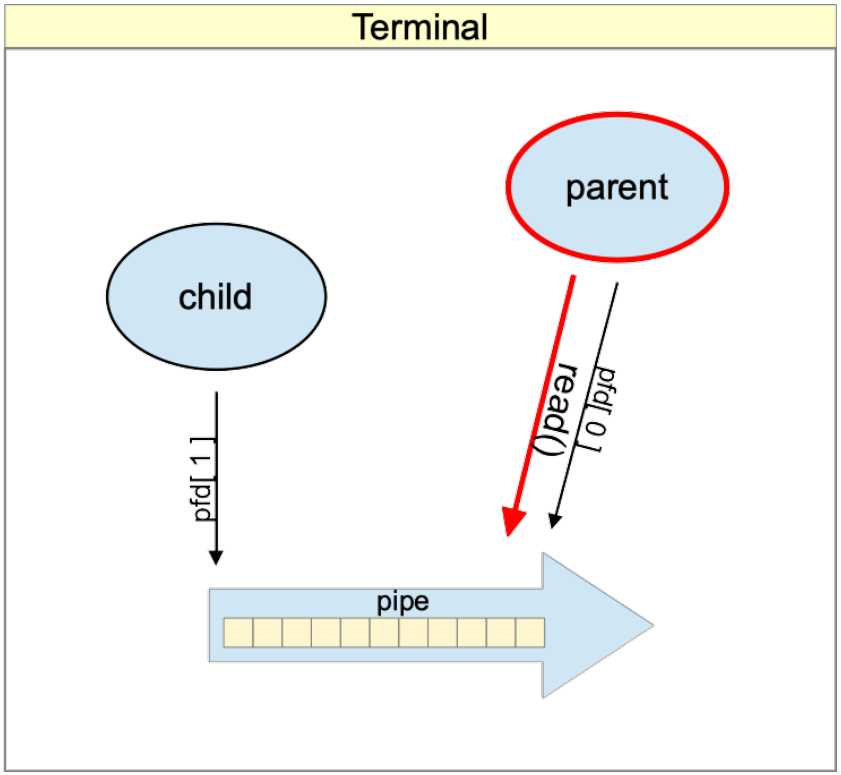
Child: Write message into Pipe



Just writing the contents of the string (not the null terminator)

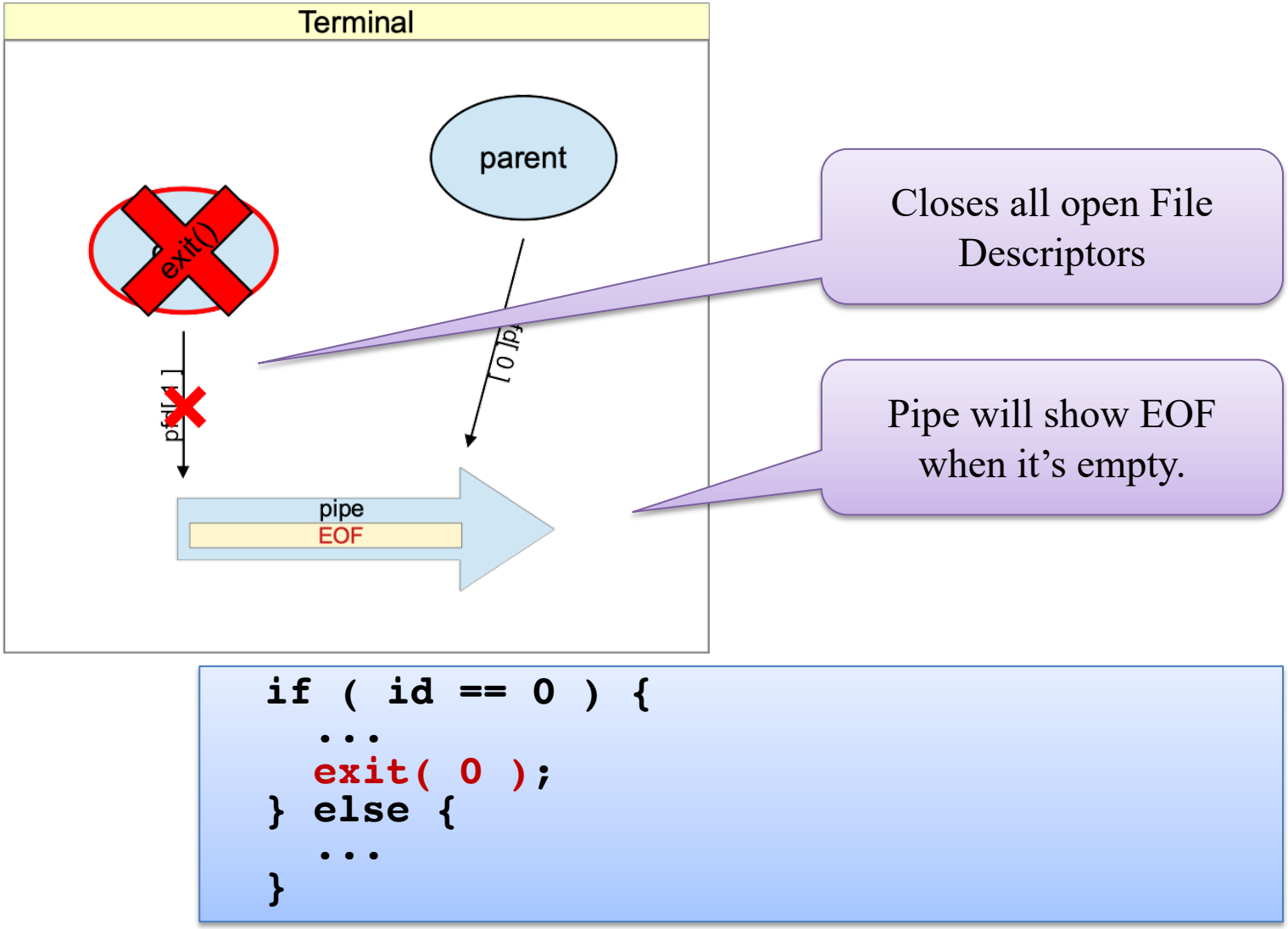
```
if ( id == 0 ) {
   ...
   int slen = strlen( buffer );
   int len = write( pfd[ 1 ], buffer, slen );
} else {
   ...
}
```

Parent: Read message from Pipe



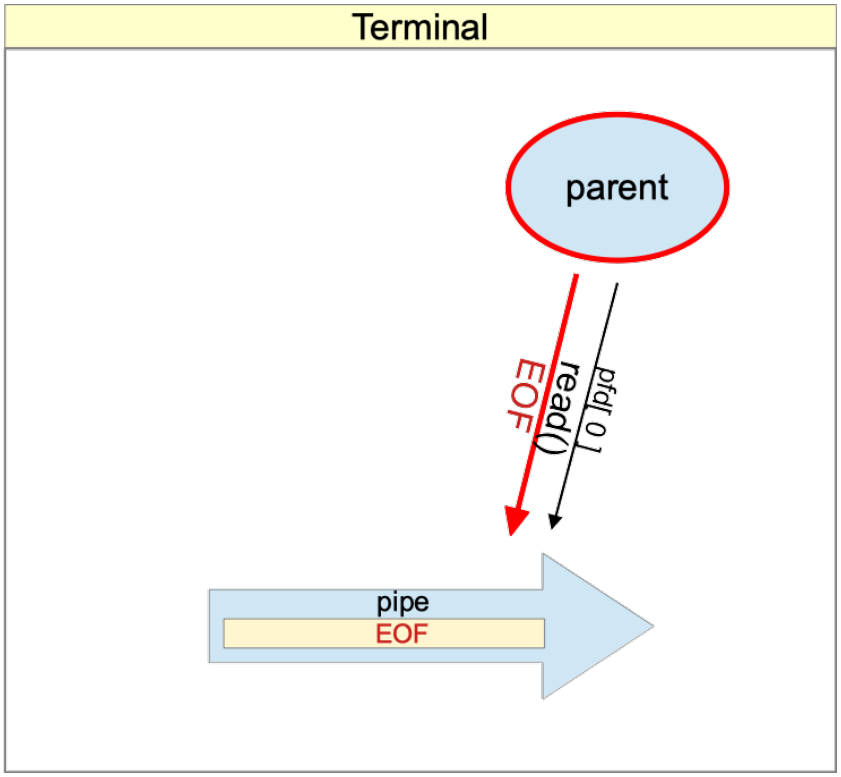

```
if ( id == 0 ) {
   ...
} else {
   ...
   char buffer[ 1024 ];
   int len = read(pfd[0], buffer, sizeof(buffer)-1);
}
```

Child: Exit after Sending Message



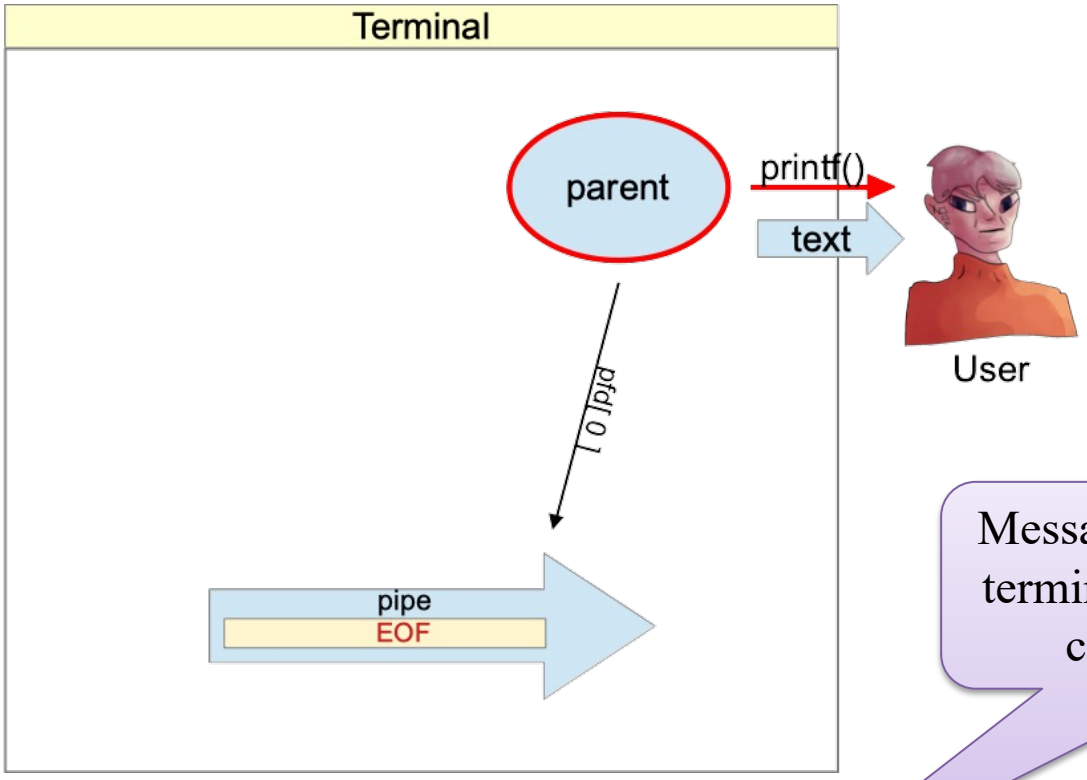

```
if ( id == 0 ) {
   ...
   exit( 0 );
} else {
   ...
}
```

Parent: Reaches EOF on Pipe Input




```
if ( id == 0 ) {
  ...
} else {
  ...
  char buffer[ 1024 ];
  int len = read(pfd[0], buffer, sizeof(buffer)-1);
}
```

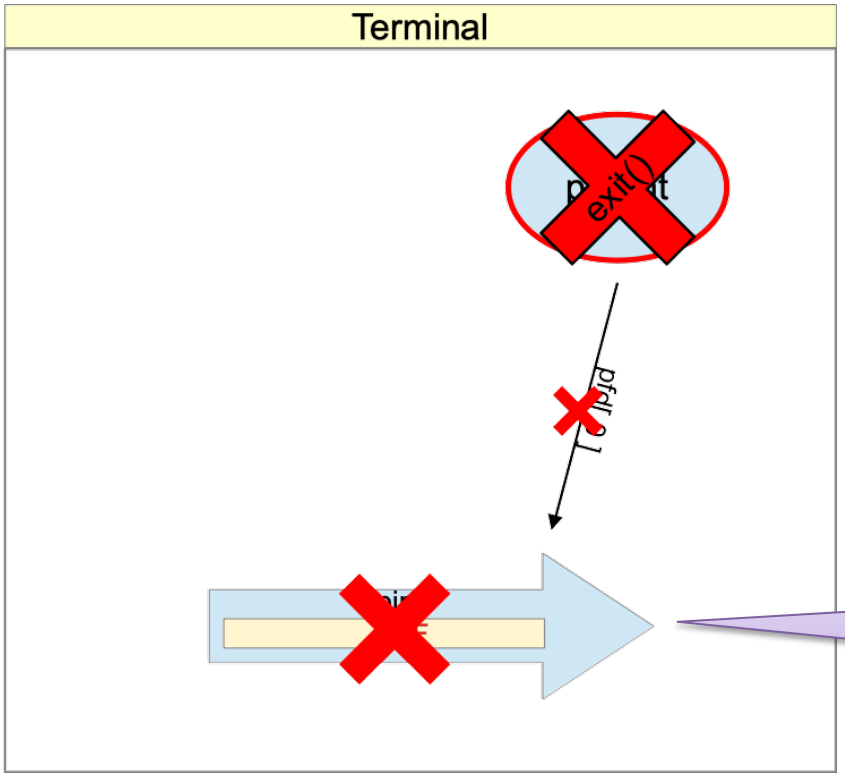
Parent: Print message for User



Message didn't include null termination … add it so we can print with **%s**.

```
if ( id == 0 ) {
  ...
} else {
  ...
  buffer[ len ] = '\0';
  printf( "Message: %s", buffer );
}
```

Parent: Exit after printing Message



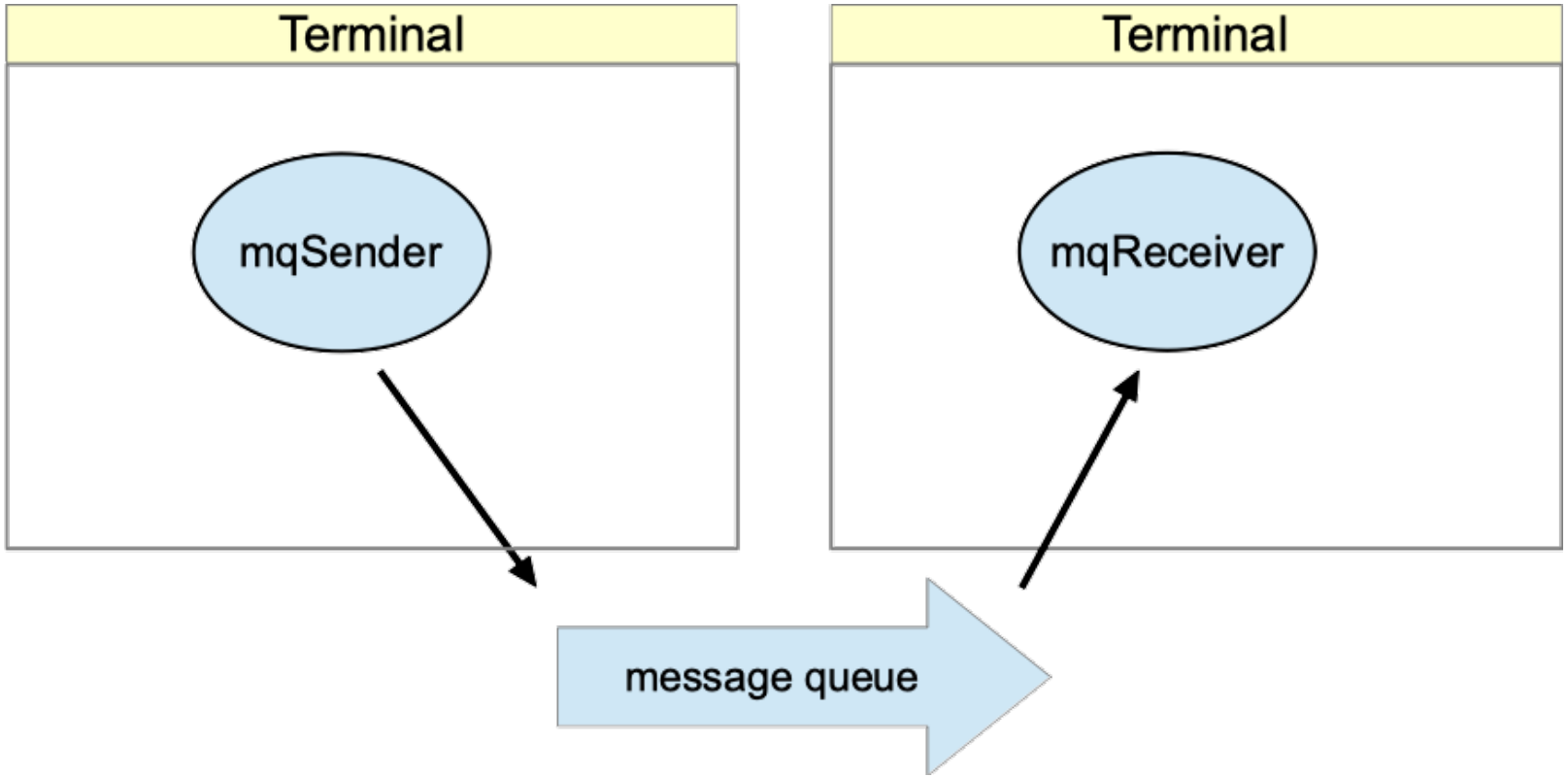
Pipe goes away when its last file descriptor is closed.

```
if ( id == 0 ) {
   ...
} else {
   ...
   return EXIT_SUCCESS;
}
```

Message Passing Example

- IPC Mechanism: Anonymous Pipes
- POSIX Calls used: pipe(), read(), write(), close()
- See, the OS is creating abstractions for its processes
  - We can use a communication channel just like a file
- Is this direct or indirect communication?
- Is there any buffering?
- How is the communication channel created?
- Are message boundaries respected?

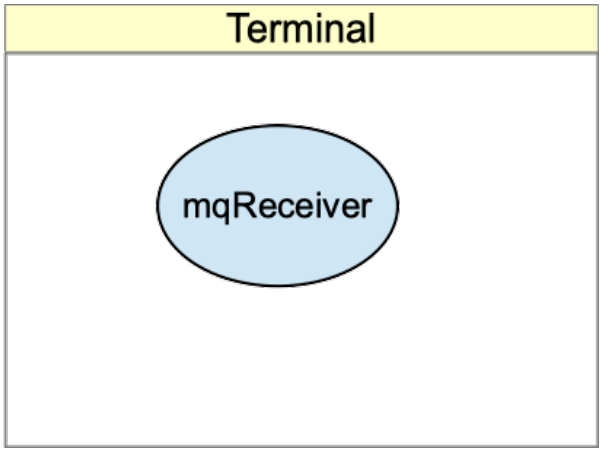
Example: POSIX Message Queues



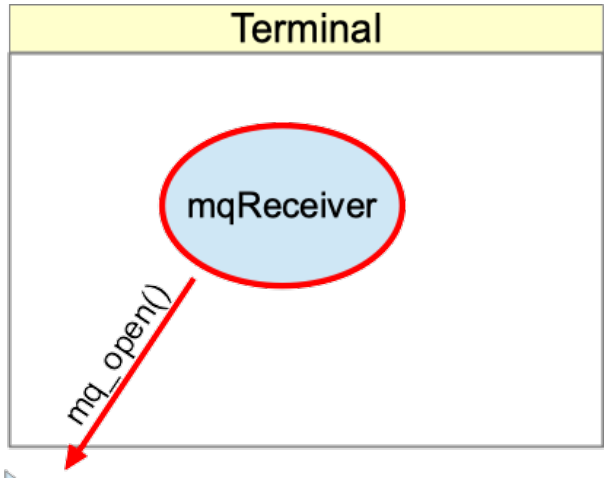
Source file: mqSender.c

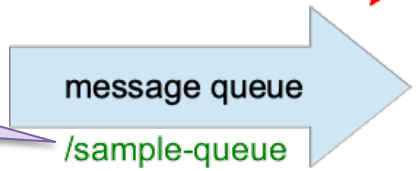
Source file: mqReceiver.c

Receiver running in a Terminal



Receiver Creates a Message Queue





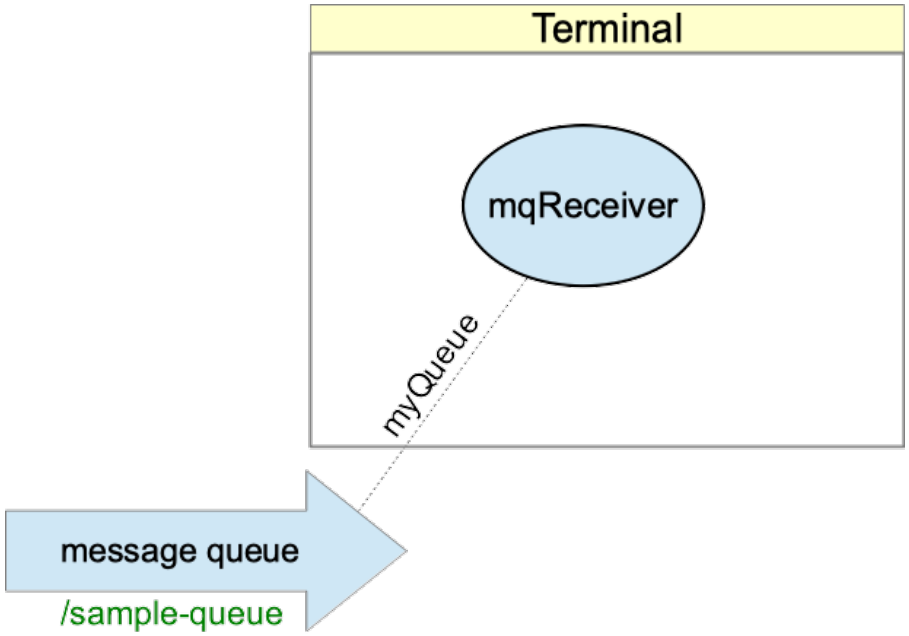
System-wide name for the queue.

Attributes for the new Message Queue

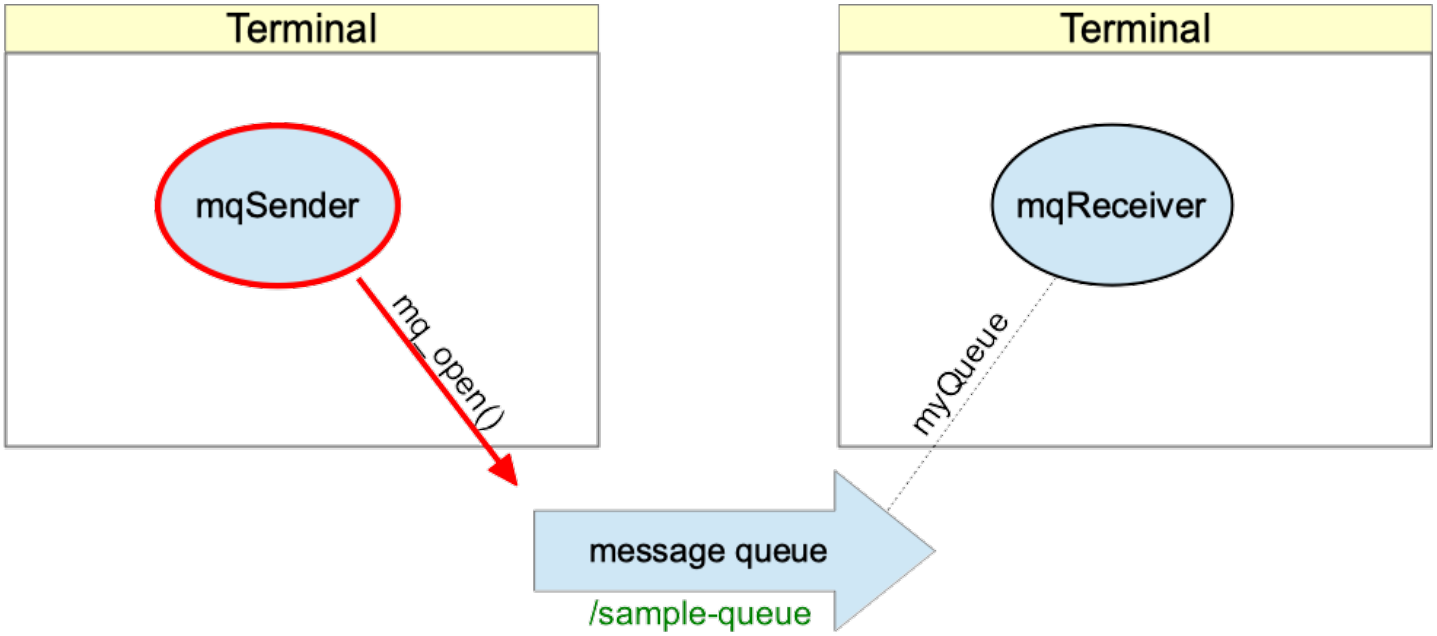
Create the message queue, but I just need to read from it.

```
#include <mqueue.h>
...
  struct mq_attr attr;
  attr.mq_flags = 0;
  attr.mq_maxmsg = 10;
  attr.mq_msgsize = MAX_SIZE;

  mqd_t myQueue = mq_open(
          "/sample-queue",
          O_RDONLY | O_CREAT,
          0600, &attr );
```

Handle for the Message Queue



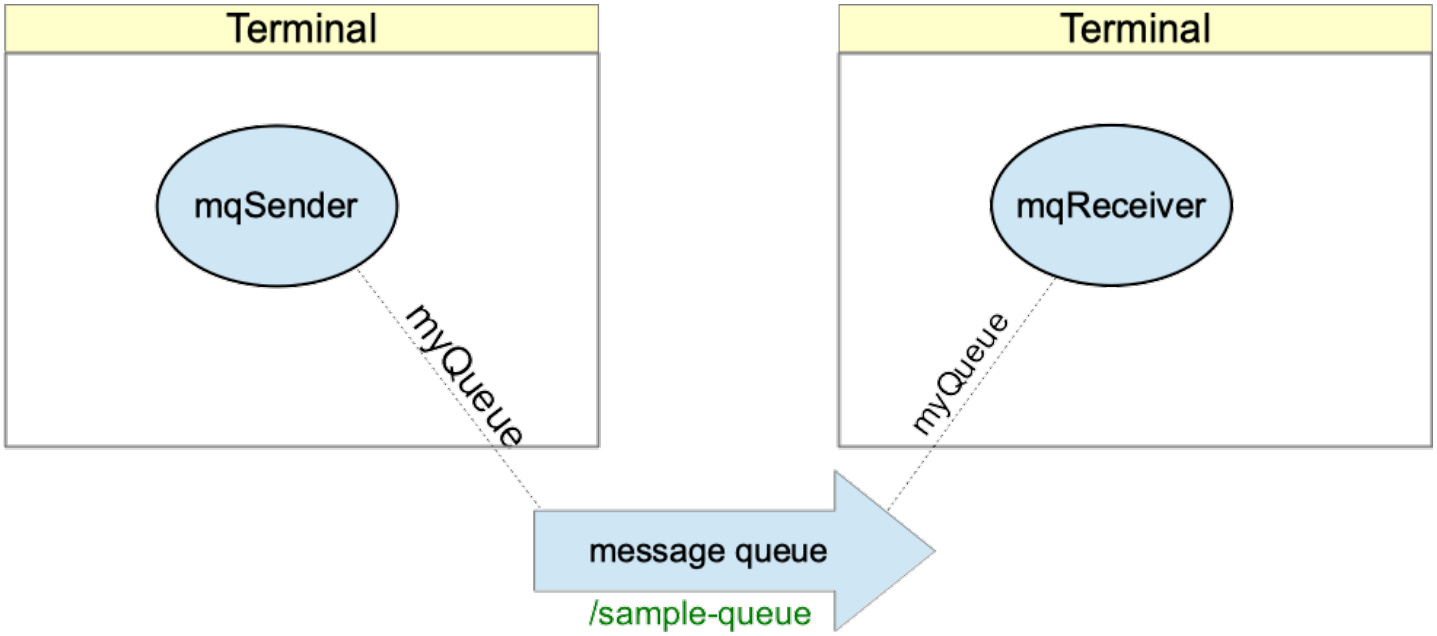
Sender Opens the Message Queue



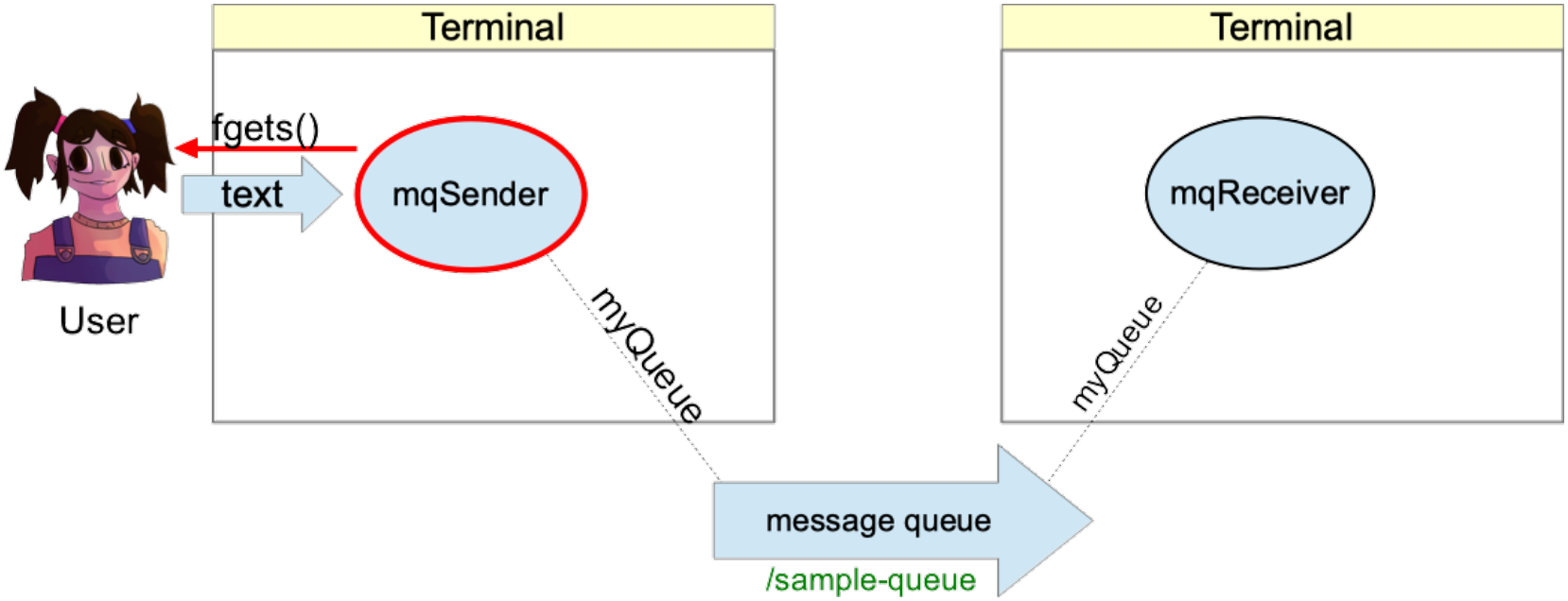

```
mqd_t myQueue = mq_open(
                  "/sample-queue",
                  O_WRONLY );
```

I don't have to create the message queue. It's already there.

Handles for the Message Queue

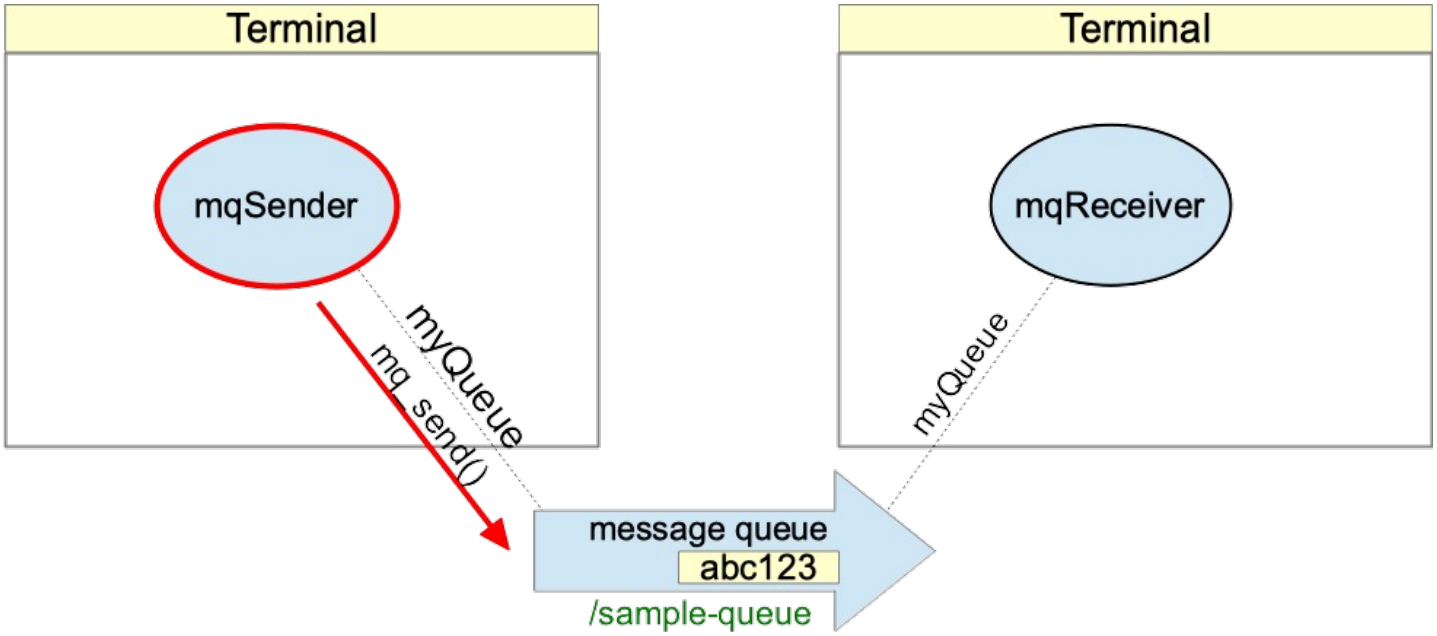


Sender gets a String from the User




```
char buffer[ MAX_SIZE + 1 ];
fgets( buffer, MAX_SIZE + 1, stdin );
```

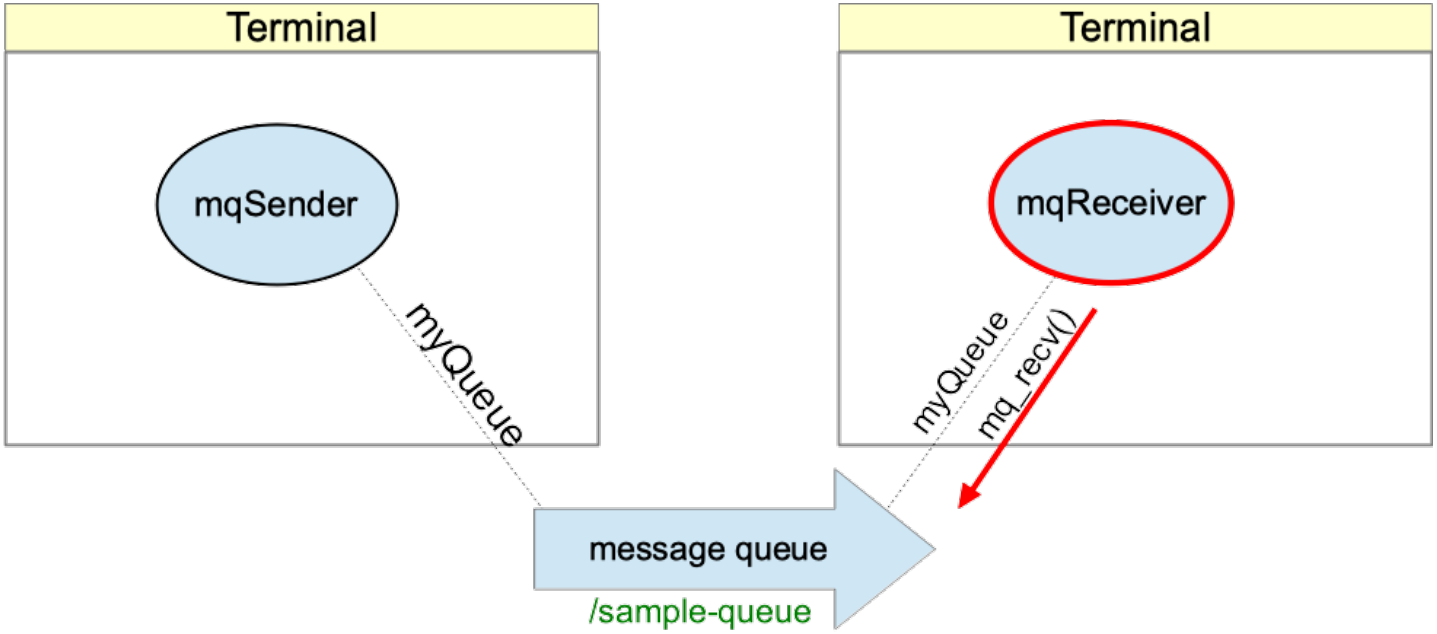
Writes String into Message Queue




```
mq_send( myQueue,
         buffer,
         strlen( buffer ),
         0 );
```

Number of bytes to send. (notice, we're not sending the null terminator)

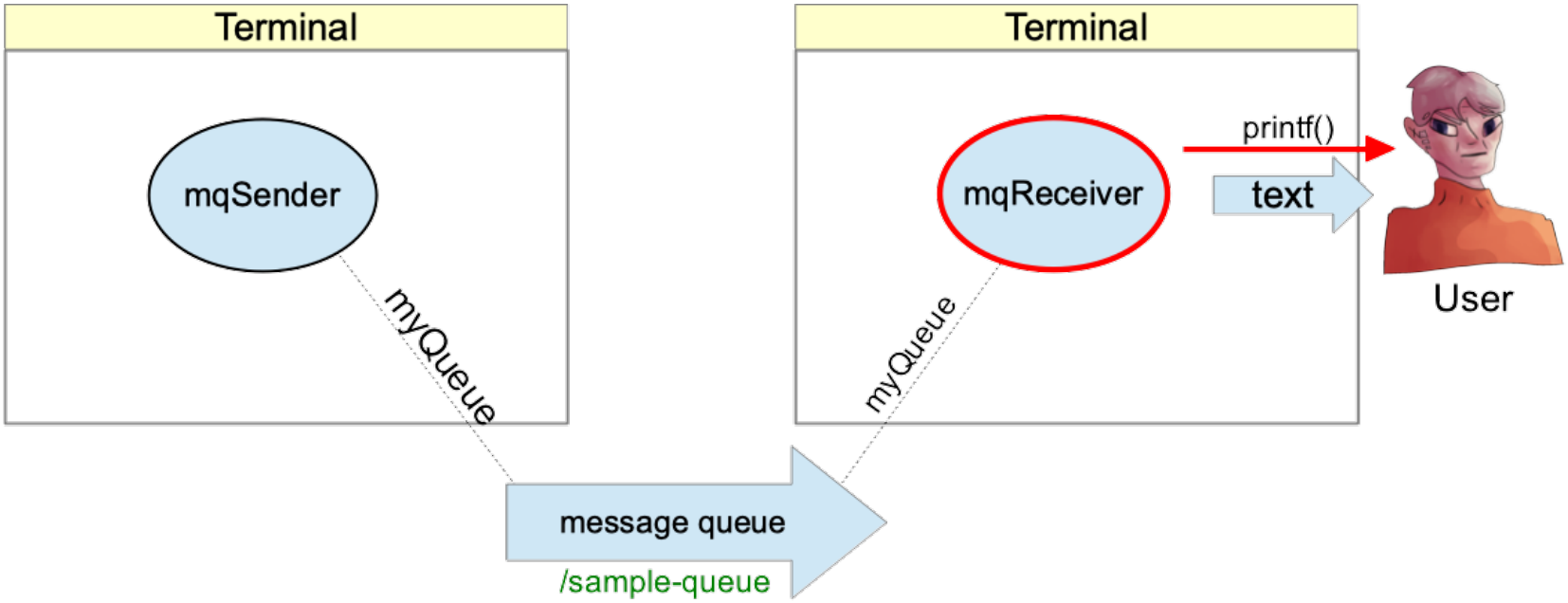
Receiver gets the String



Return value tells us the message length.

```
int len = mq_receive( myQueue,
                      buffer,
                      sizeof(buffer),
                      NULL );
```

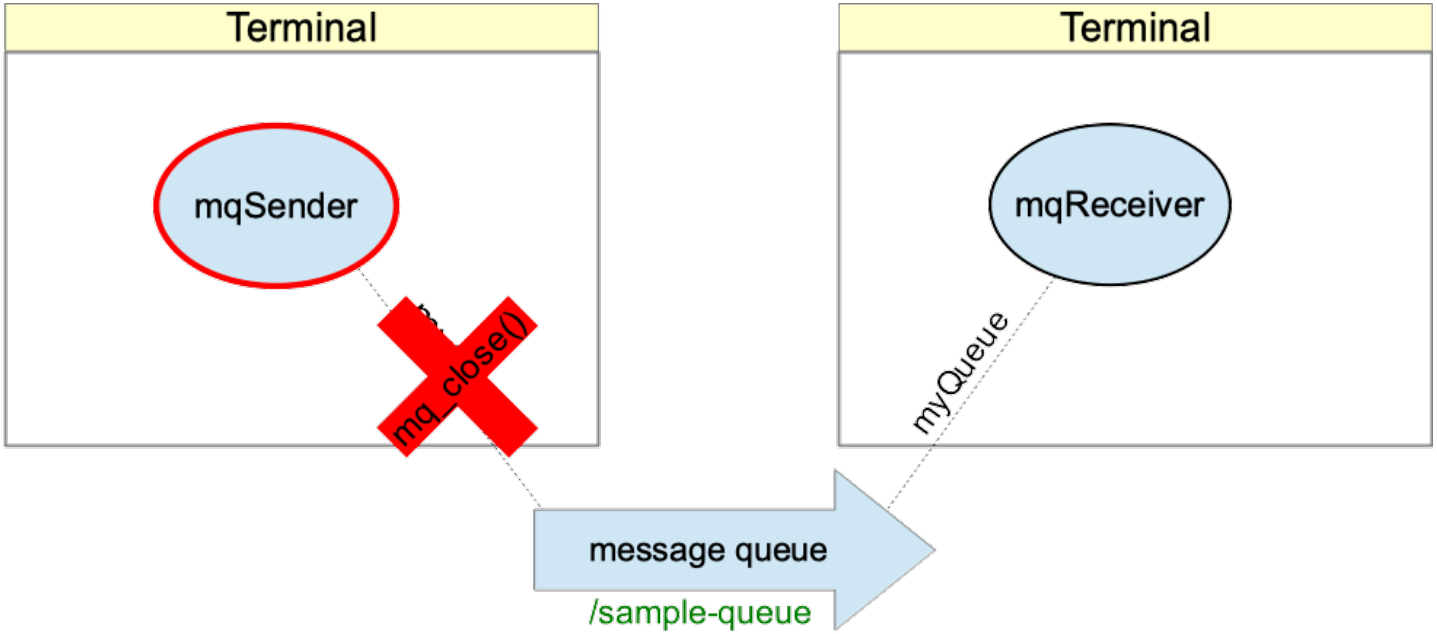
Receiver prints the String



No null terminator; receiver just prints the message one character at a time.

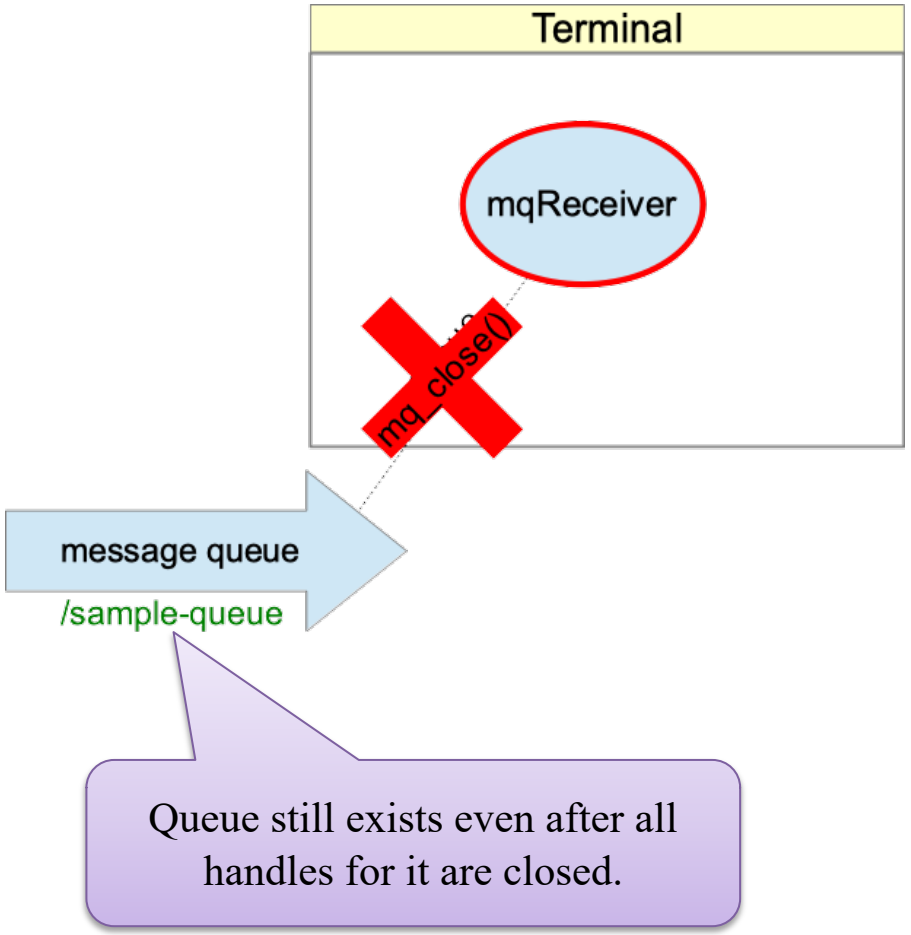
```
for ( int i = 0; i < len; i++ )
    printf( "%c", buffer[ i ] );
```

Sender Closes Message Queue and Exits



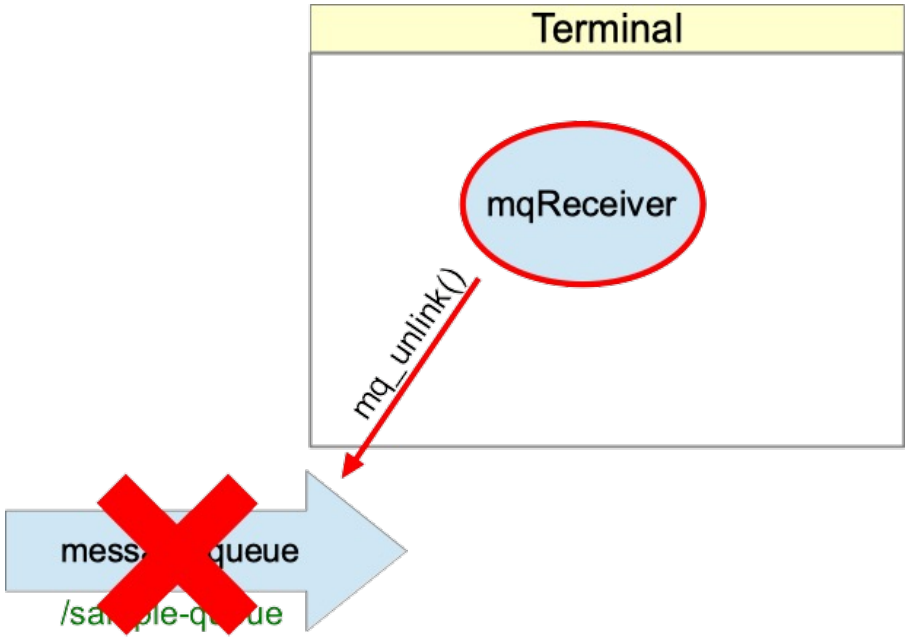

```
mq_close( myQueue );
```

Receiver Closes Message Queue




```
mq_close( myQueue );
```

Receiver Destroys the Message Queue




```
mq_unlink( "/sample-queue" );
```

Message Passing Example

- IPC Mechanism: Message queues
- POSIX Calls used: mq_open(), mq_send(), mq_receive(), mq_close(), mq_unlink()
- Direct or Indirect?
- Buffering?
- Are message boundaries respected?

Blocking

- What if we call read() and there's no message?
  - Normally, we expect the process to just wait
  - How? Just like it does when it waits for I/O
  - That's called *blocking*
  - Process goes into the waiting state when it can't make progress
- Blocking is an important mechanism provided by the OS
  - Implemented as another queue of PCBs

Blocking vs. Non-Blocking

- OS may offer blocking and non-blocking interfaces
  - Blocking, the process waits until its request is done
  - Non-blocking, typically the process has to check back later
    - i.e., the process can perform polling
    - Or, the OS could notify it via an upcall
    - (or, there are system calls to check for I/O on multiple file descriptors at once)

Blocking in IPC

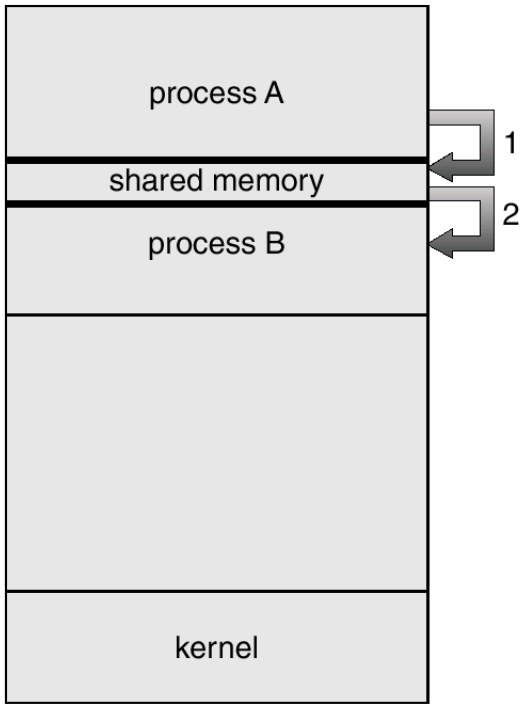
- *Blocking Receive*: Must get a message before continuing
- *Non-blocking Receive*: check for a message, check back later
- *Blocking Send*: wait until you can get rid of the message (either to the OS or the receiver)
- *Non-blocking Send*: try to send, try again later

Buffering

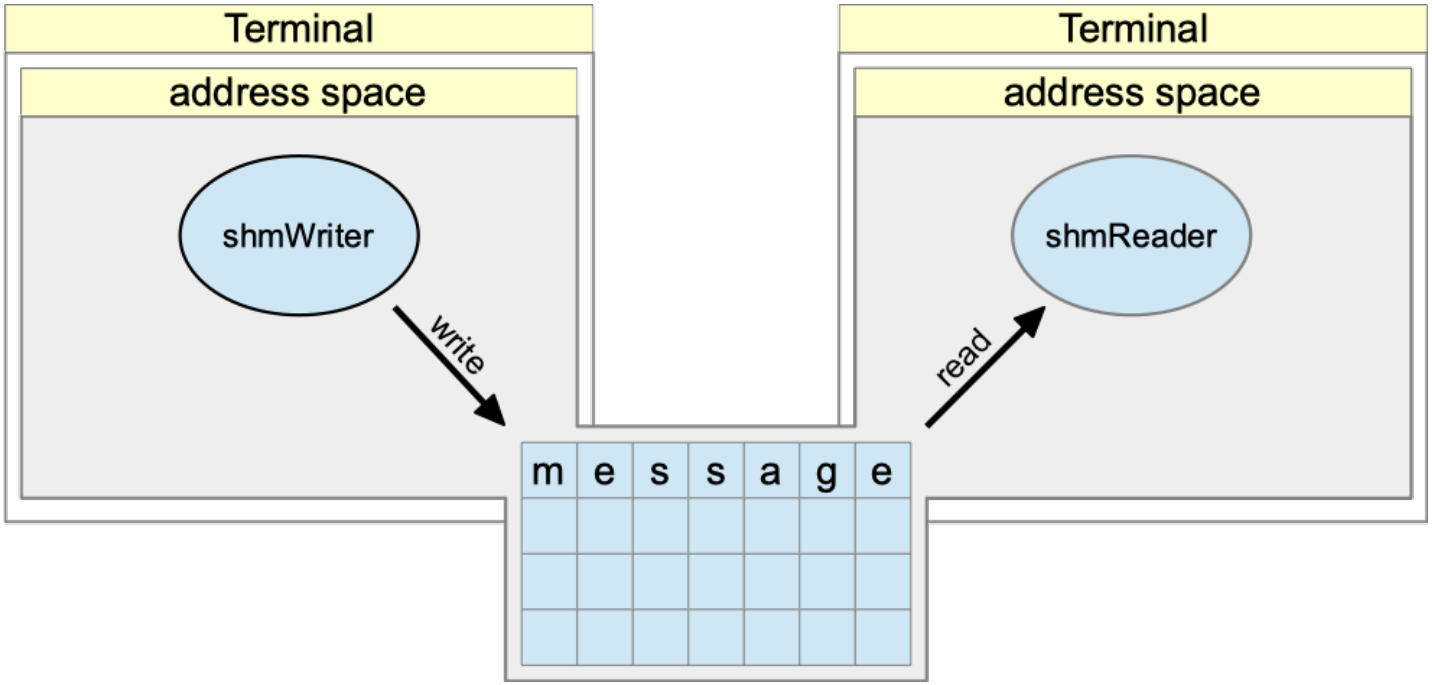
- OS can hold messages temporarily
  - Messages that have been sent but not yet received
  - Let the sender make progress even if receiver falls behind
- Buffer capacity
  - Zero capacity, with blocking → rendezvous
  - Bounded capacity
  - Unbounded capacity → sender never has to wait

Shared Memory Communication

- Two programs connected via shared memory
  - Write into shared memory to:
    - Communicate
    - Or, just update a shared data structure
  - Read from shared memory to:
    - Receive a message
    - Examine a shared data structure
- Don't need explicit calls for communication



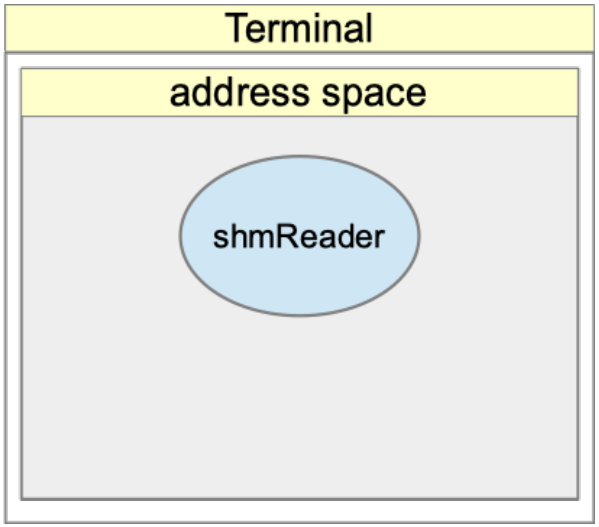
Example: Shared Memory



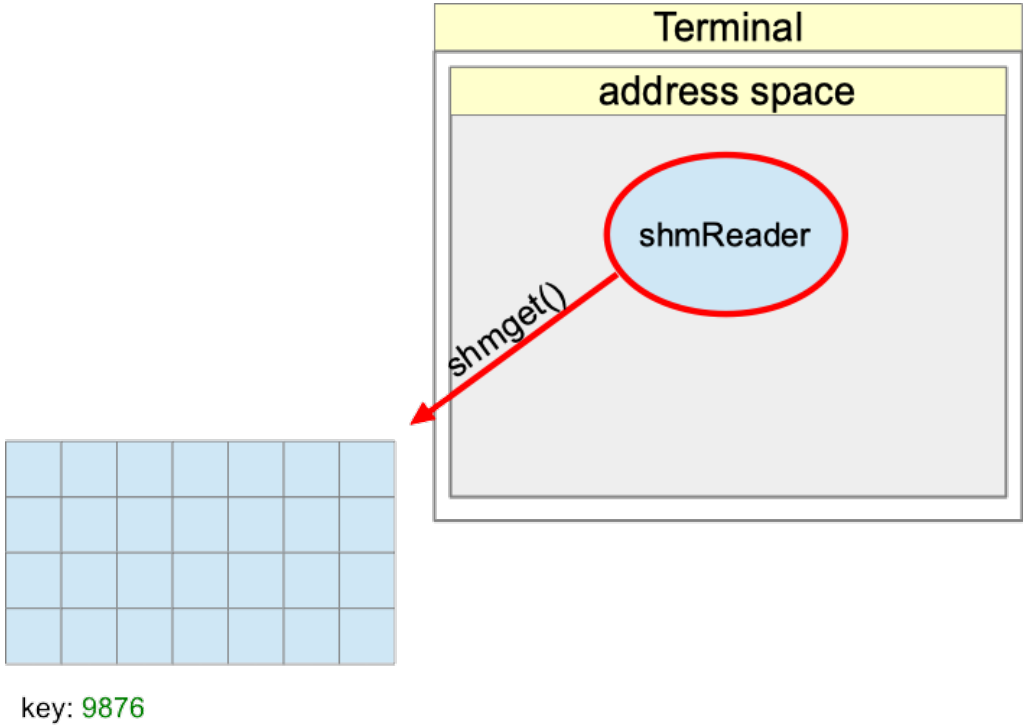
Source file: shmWriter.c

Source file: shmReader.c

Reader: Running in a Terminal



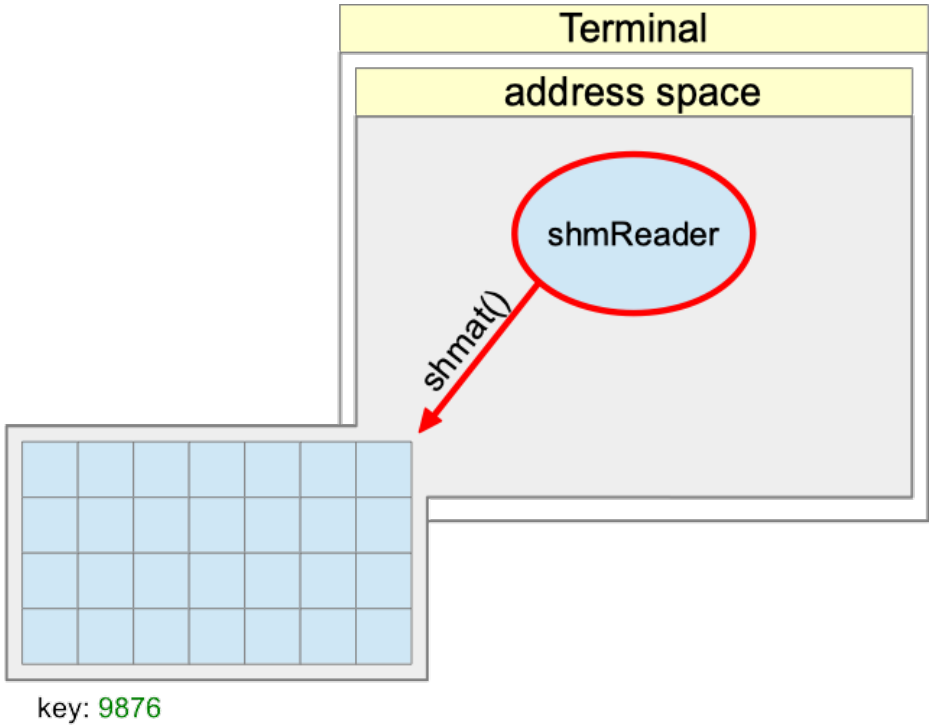
Reader Creates a Shared Memory Segment



Name of shared memory (key)
Size of memory
Flags and permissions

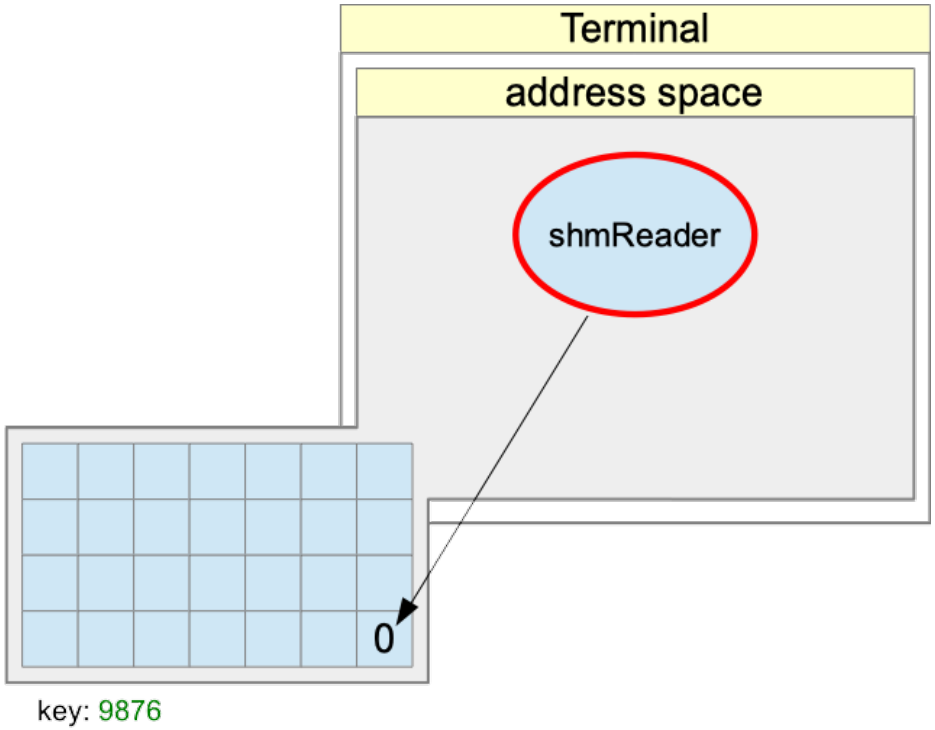
```
int shmid = shmget(9876,
                   BLOCK_SIZE,
                   0666 | IPC_CREAT);
```

Reader adds Shared Memory to Address Space



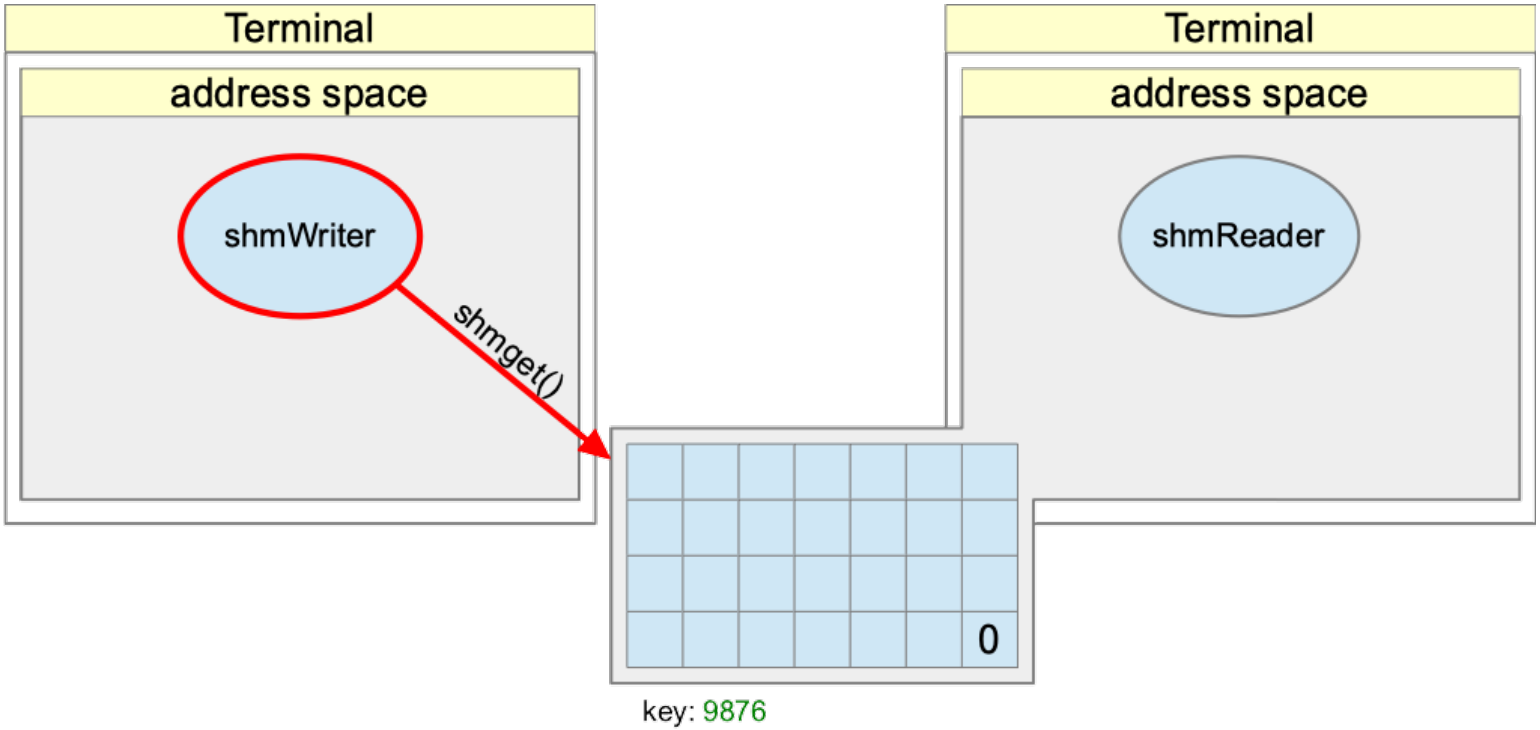

```
char *sbuffer = (char *)shmat(shmid,
                              0,
                              0);
```

Setting a Flag for Message Detection




```
sbuffer[ BLOCK_SIZE - 1 ] = 0;
```

Writer gets a Handle for the Shared Memory

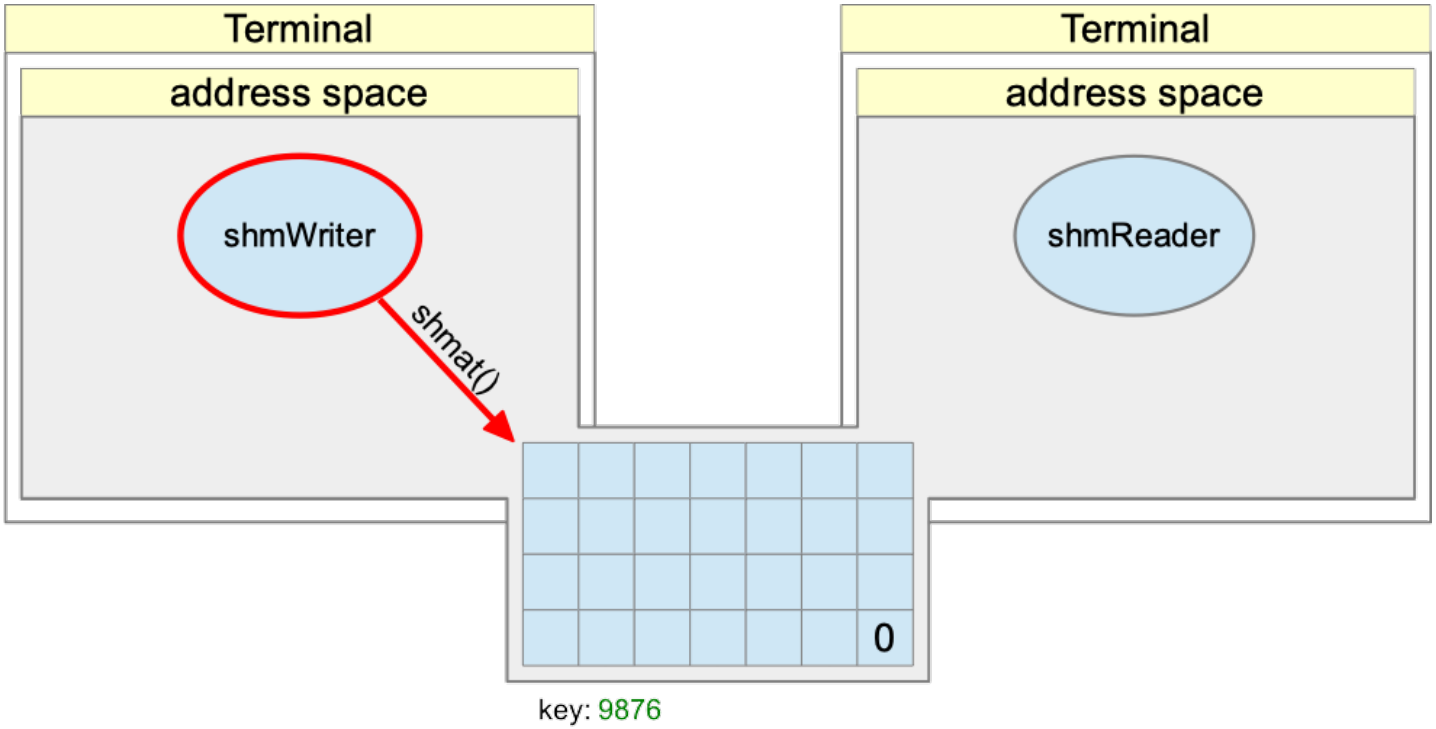


Same key, same size.

Writer doesn't need to create a new segment.

```
int shmid = shmget( 9876,
                    BLOCK_SIZE,
                    0 );
```

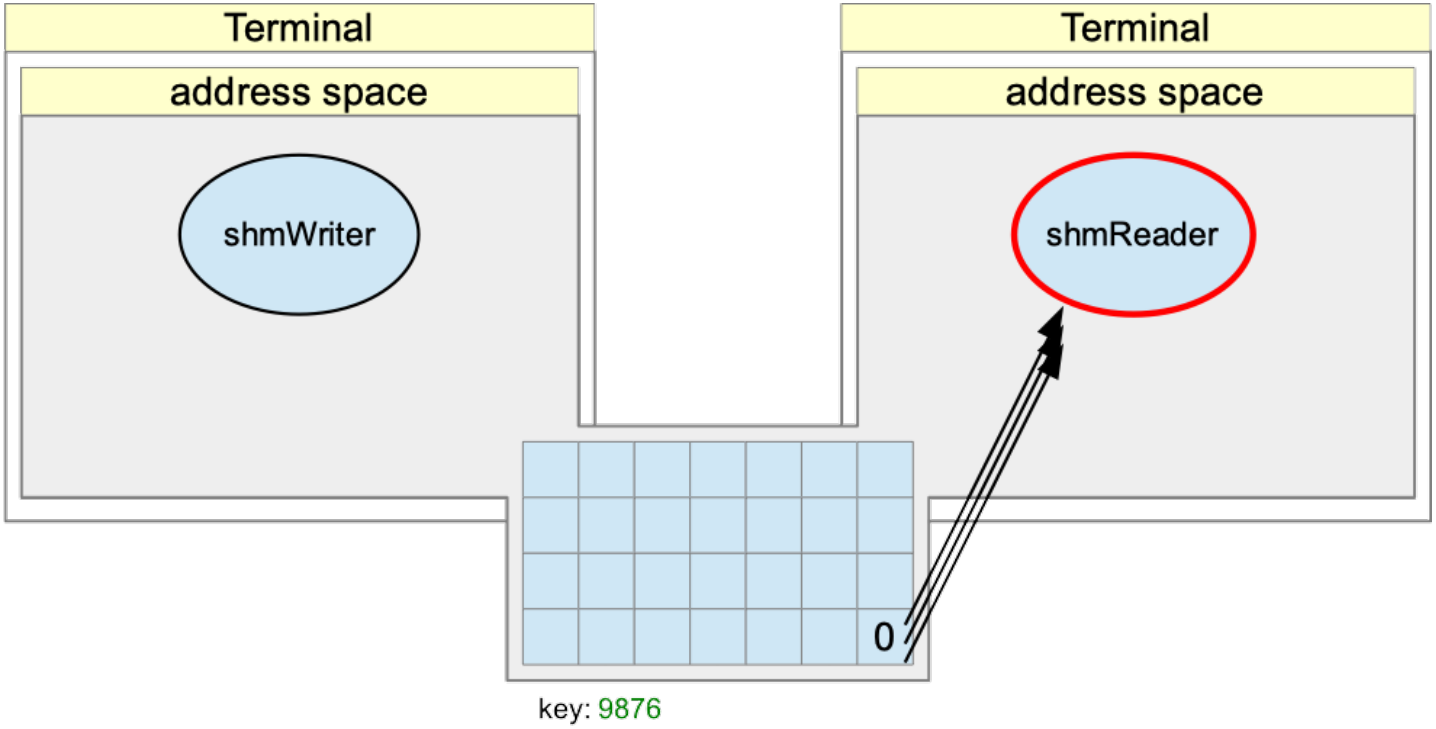
Writer adds Shared Memory to Address Space




```
char *sbuffer = (char *)shmat( shmid,
                               0,
                               0 );
```

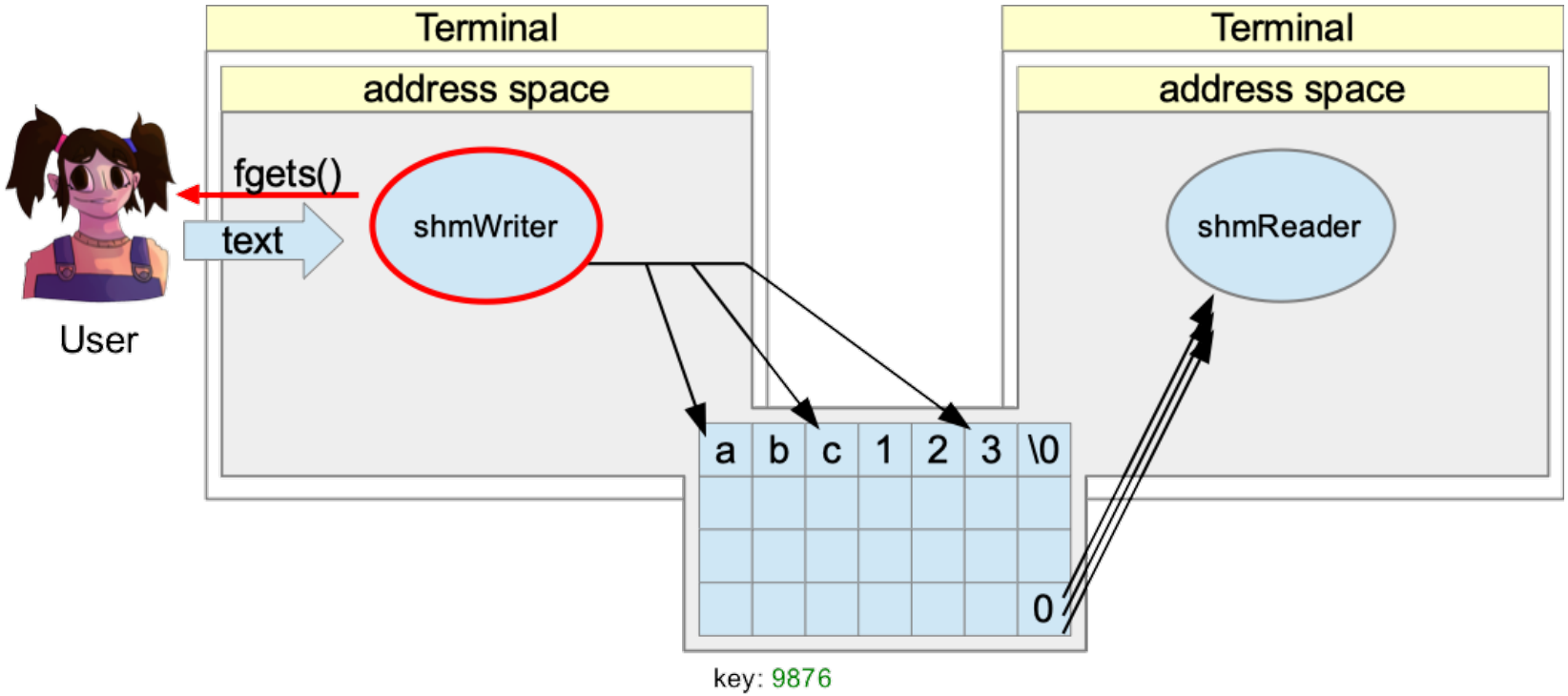
Reader Polls for a Message

(busy waiting, this part is bad)




```
while (sbuffer[BLOCK_SIZE - 1] == 0)
    ;
```

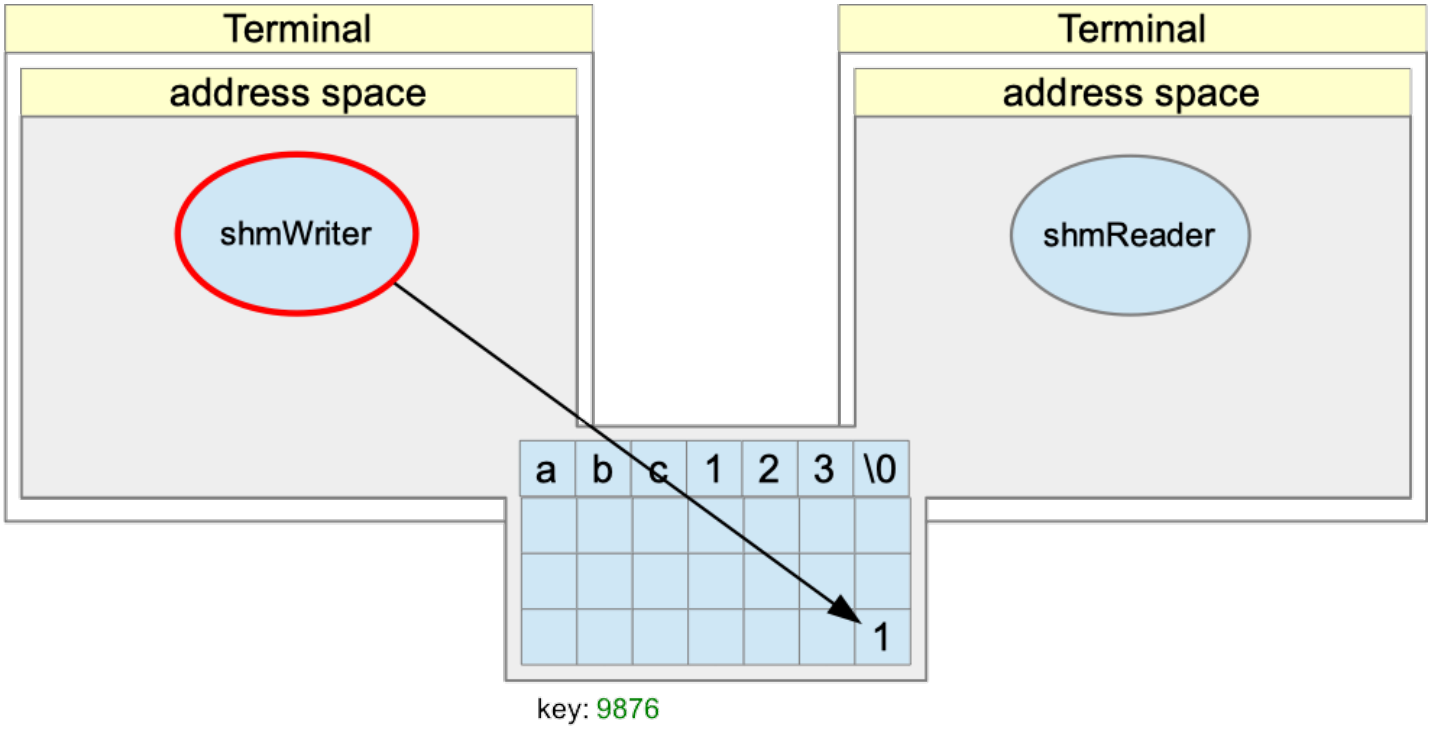
Writer gets a Message



Message read directly into shared memory. It' like any other part of memory.

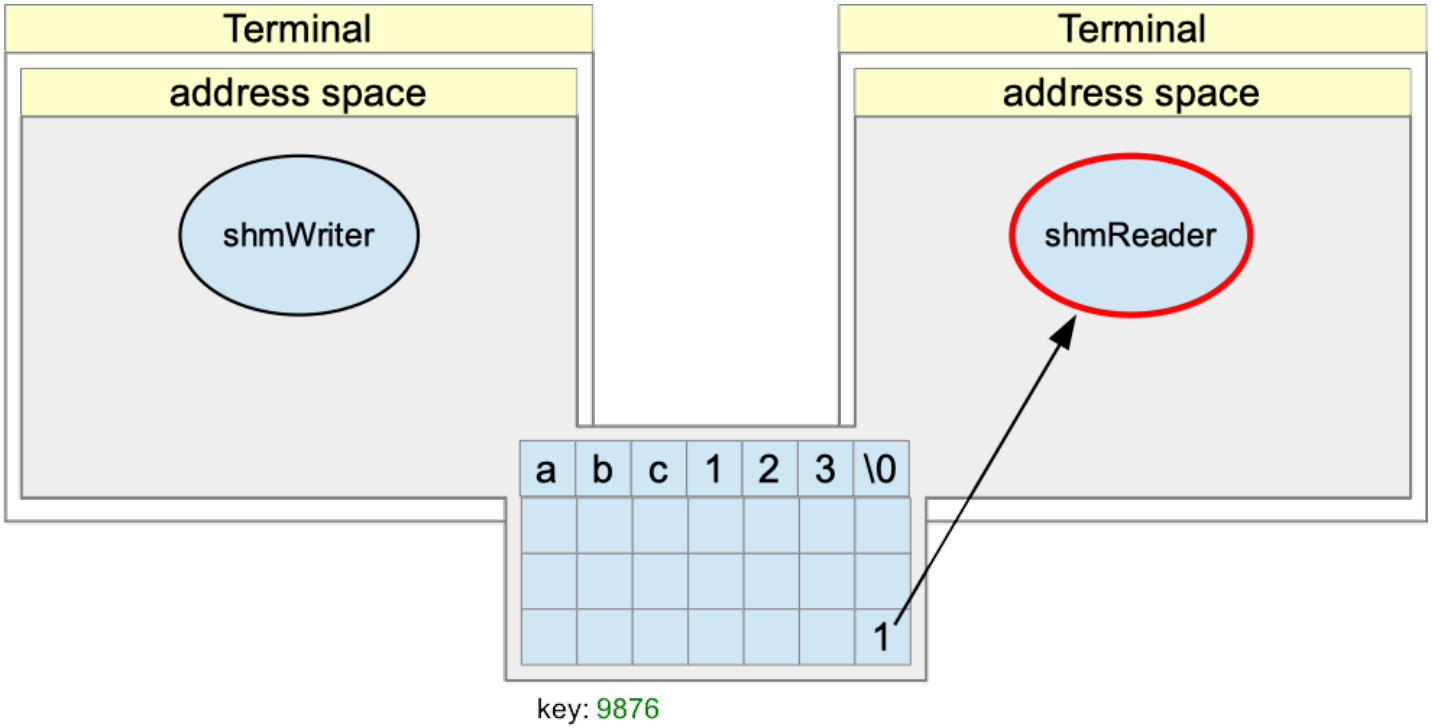
```
fgets( sbuffer,
       BLOCK_SIZE - 1,
       stdin );
```

Writer sets the Flag



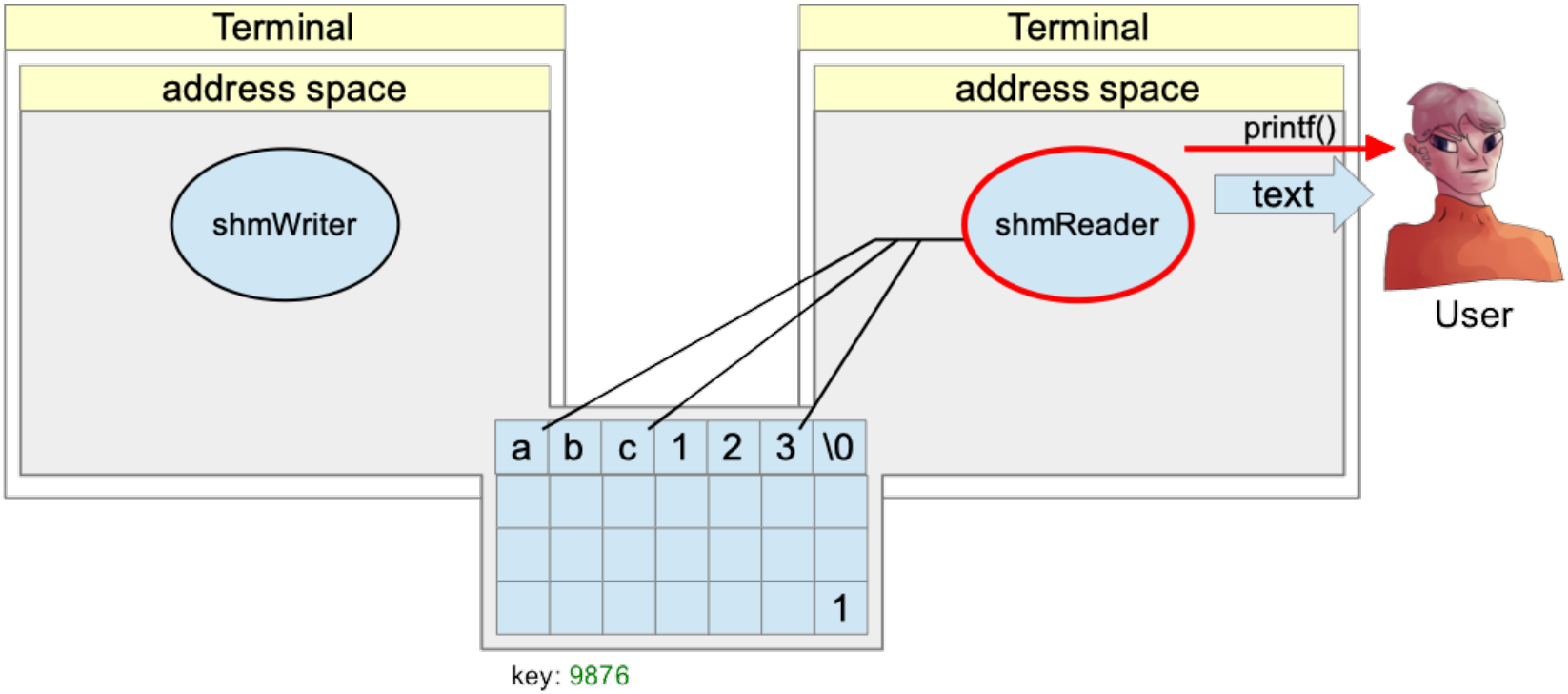

```
sbuffer[ BLOCK_SIZE - 1 ] = 1;
```

Reader notices the non-zero Flag




```
while (sbuffer[BLOCK_SIZE - 1] == 0)
    ;
```

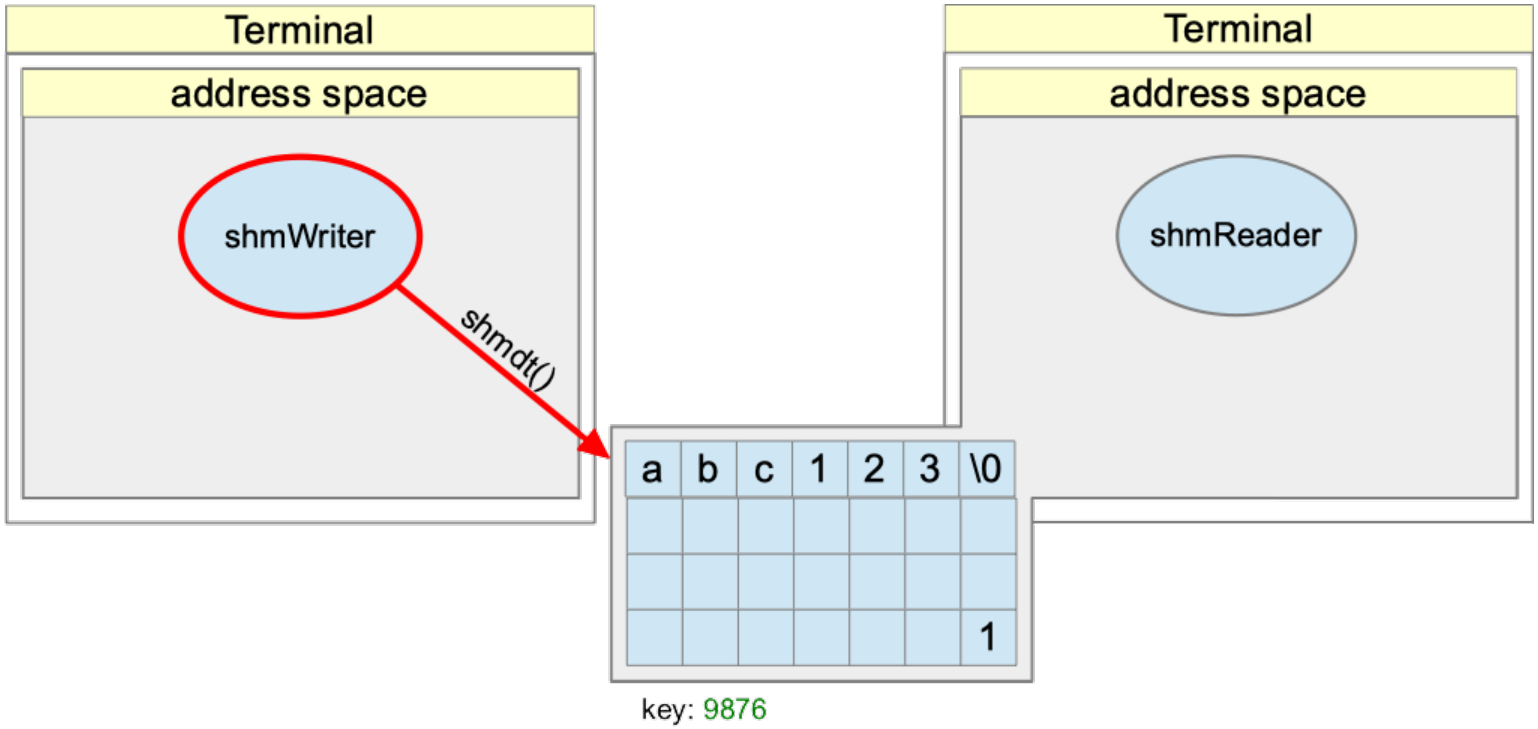
Reader prints the Message



Printing directly from the shared memory.

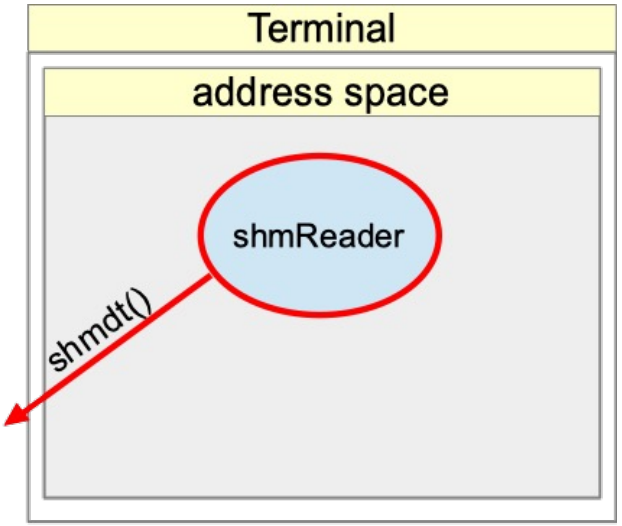
```
printf( "%s", sbuffer );
```

Writer Detaches Shared Memory and Exits




```
shmdt( sbuffer );
exit( EXIT_SUCCESS );
```

Reader Detaches Shared Memory



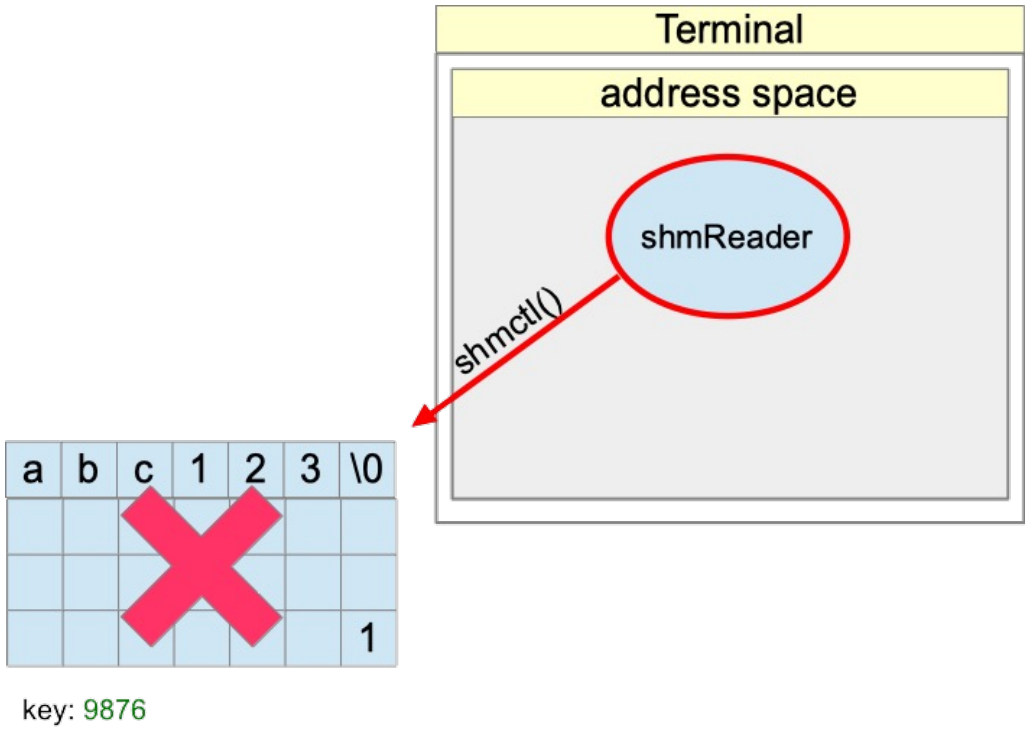
a	b	c	1	2	3	\0
						1

key: 9876

Shared memory remains even after all clients have stopped using it.

```
shmdt( sbuffer );
```

Reader Deletes Shared Memory




```
shmctl( shmid, IPC_RMID, 0 );
exit( EXIT_SUCCESS );
```

Shared Memory Example

- POSIX calls used: shmget(), shmat(), shmdt(), shmctl()
- Is this direct or indirect?
- *Busy waiting* until a message is available
  - This is bad, why?
  - Really, this is just a poor *synchronization* solution
- Would be good if the OS could help with this
  - Then, a process could block when it needed to wait for something in another process.
  - So, with shared memory we have no explicit calls to send/recv, but …
  - … typically, we still need calls for synchronization

What We Learned

- A Process and how it's implemented
- Context Switch, Process Control Block related OS structures (process table, scheduling queues, etc.)
- System calls supporting process creating and termination
- IPC via message passing and shared memory
- Blocking and Buffering