

Processes

CS 3113

Processes

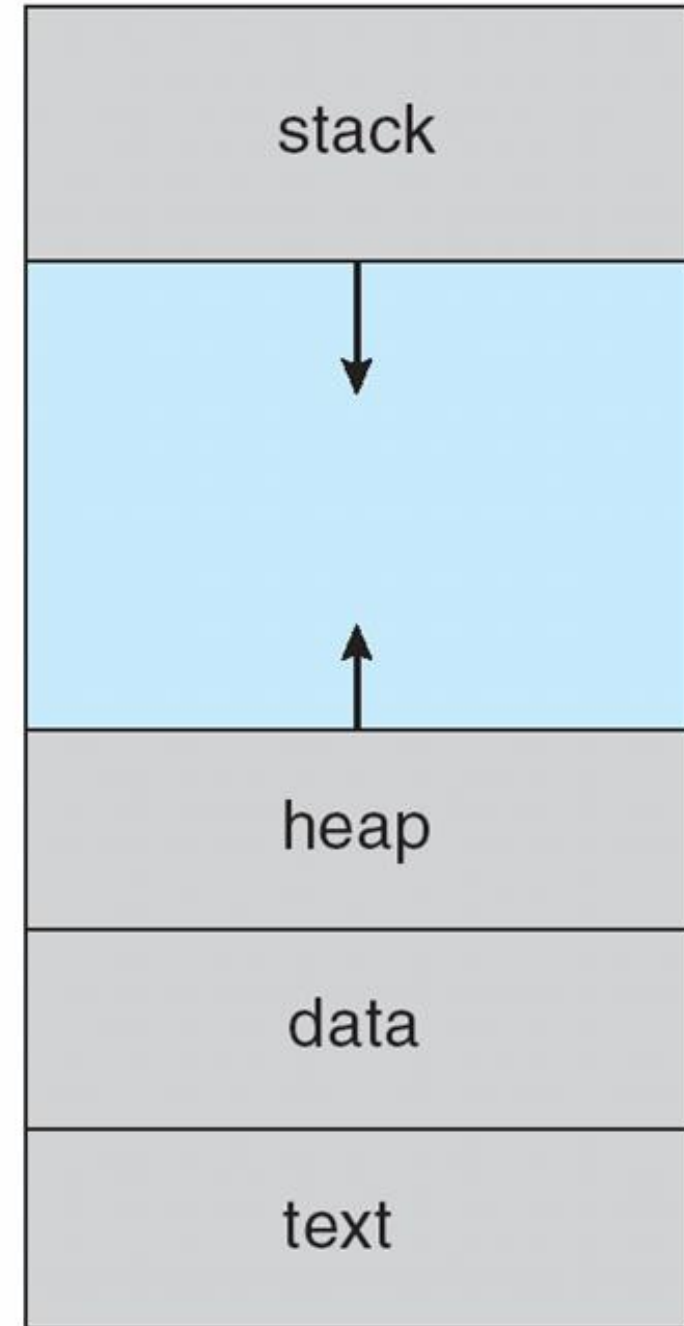
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
 - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
 - Consider multiple users executing the same program

Processes and Memory

On process creation, the process is effectively given its own memory space

- Text: storage of code
- Data: global variables (preallocated space)
- Heap: dynamically allocated space
- Stack: local variable storage

max



Stack and Heap

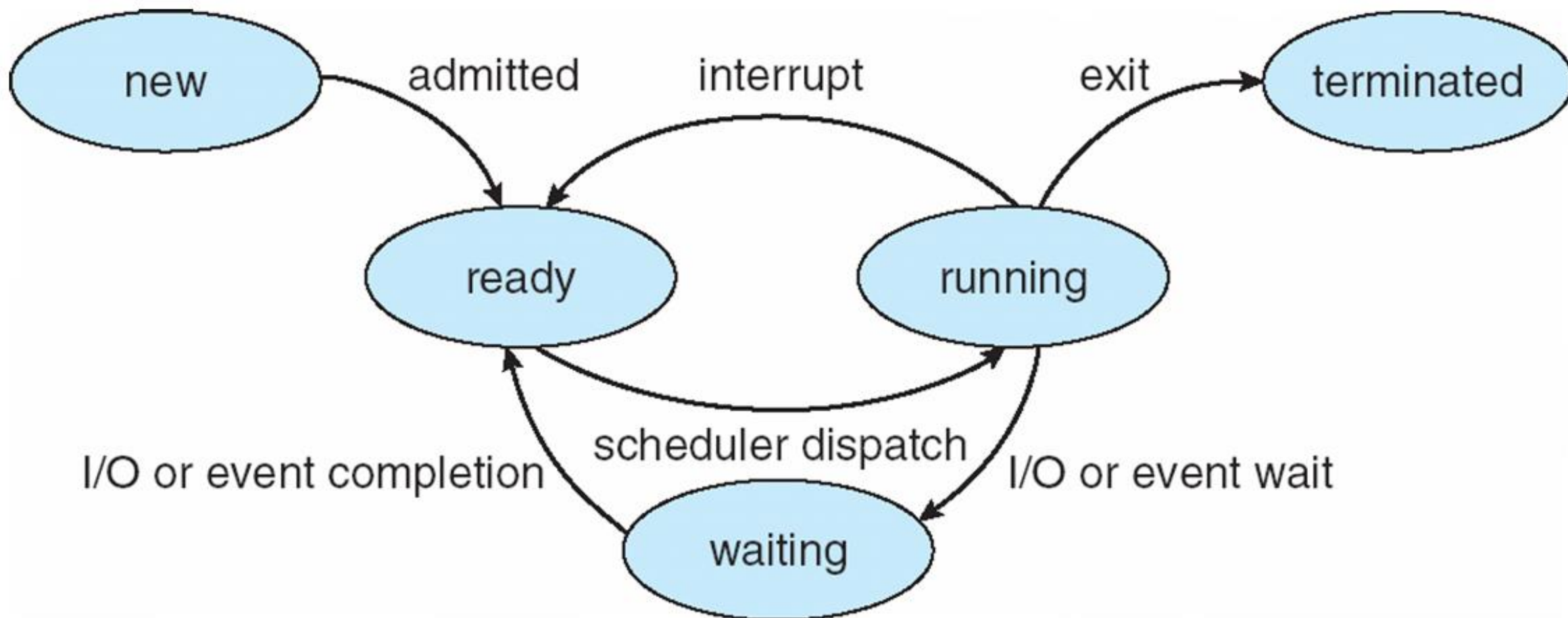
- Stack grows downward with each nested function call
 - Local variables, register state, return memory address
- Heap
 - Storage of dynamically allocated items that must be persistent across function calls (and returns from function calls)
 - OOP languages: object instantiation is done in the heap

Process State

A process is in exactly one state at any instant in time:

- **new**: The process is being created
- **running**: Instructions are being executed by the CPU
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

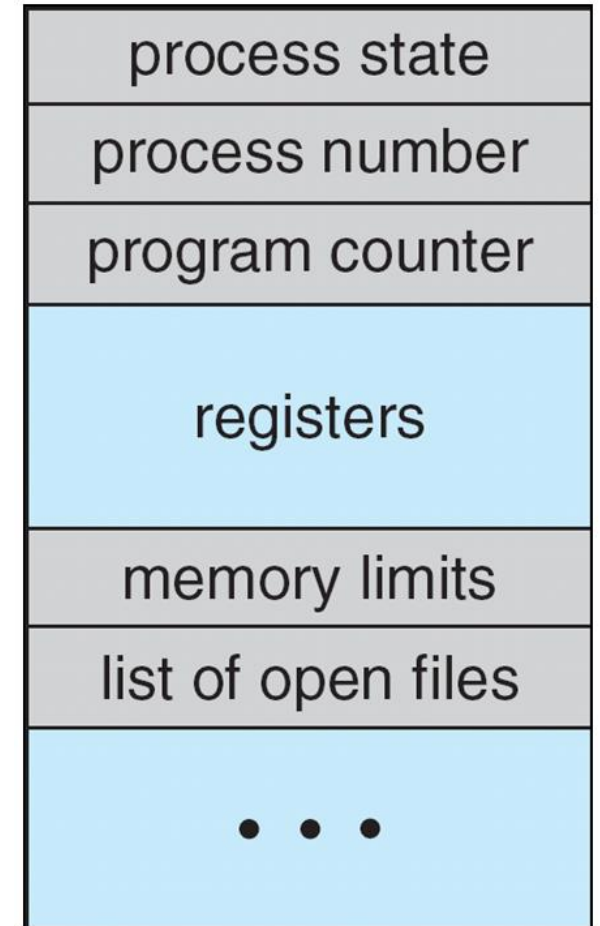
Process State



Kernel Data Structure: Process Control Block

Stores information about the running process:

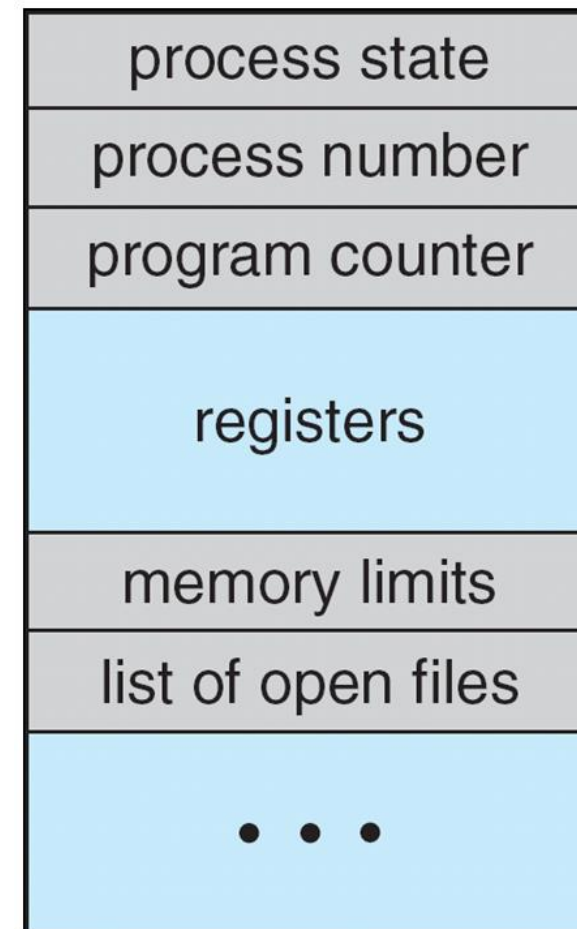
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers



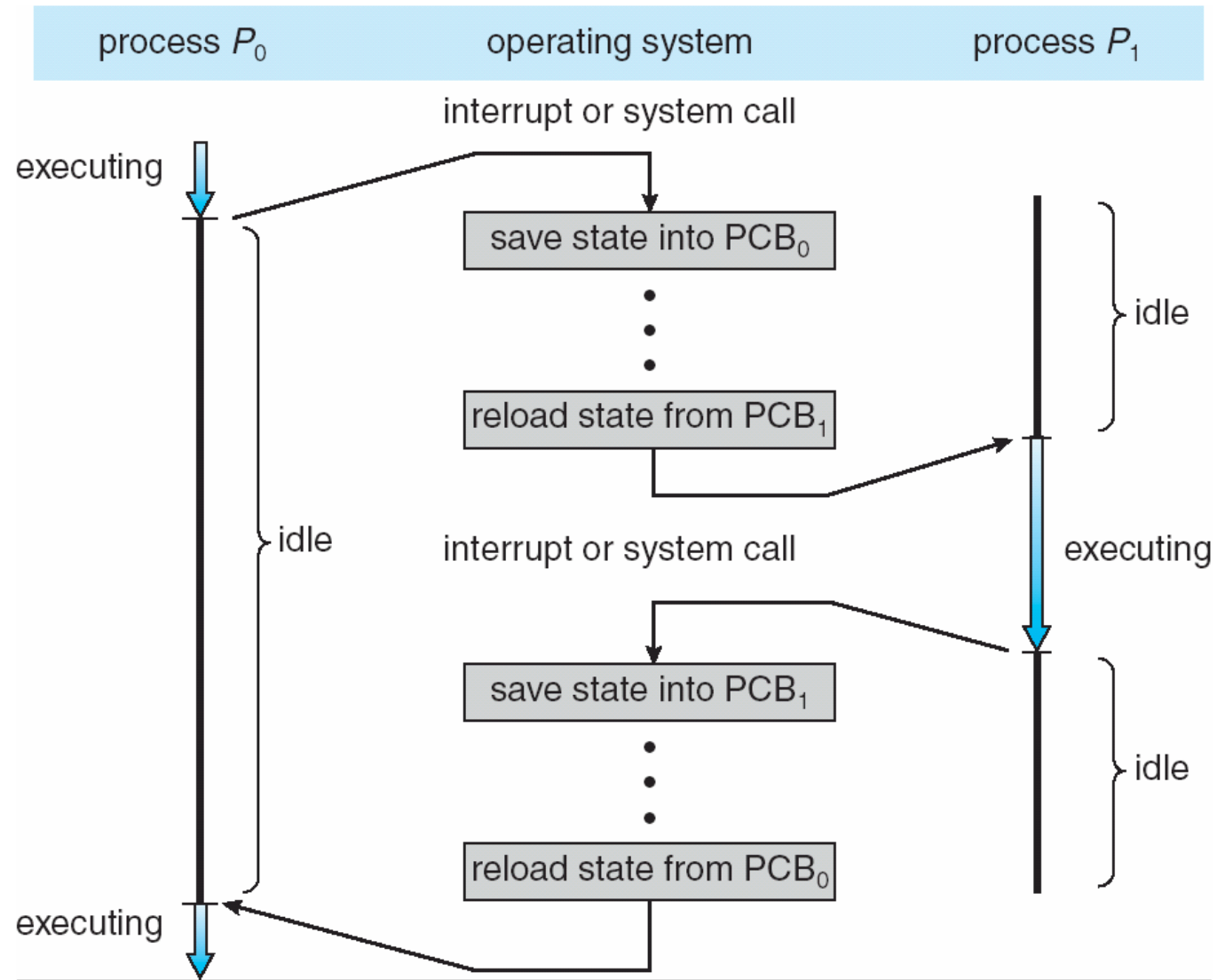
Kernel Data Structure: Process Control Block

continued:

- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



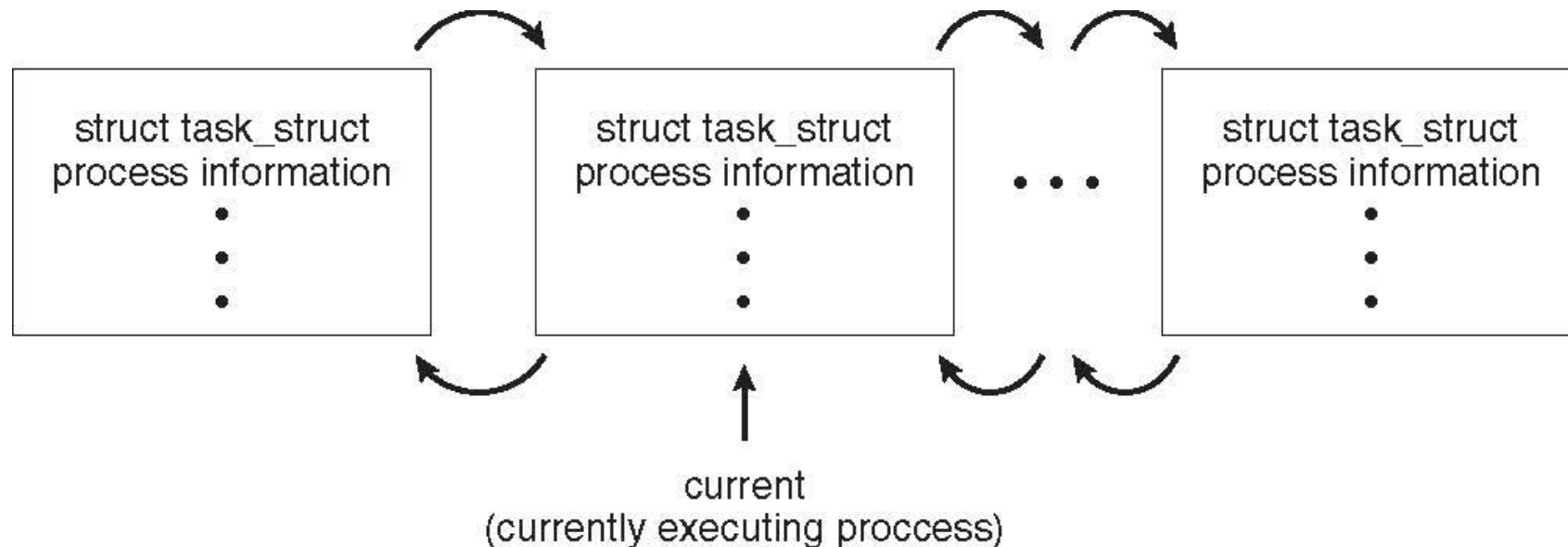
CPU Switching from One Process to Another



Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Process Scheduling

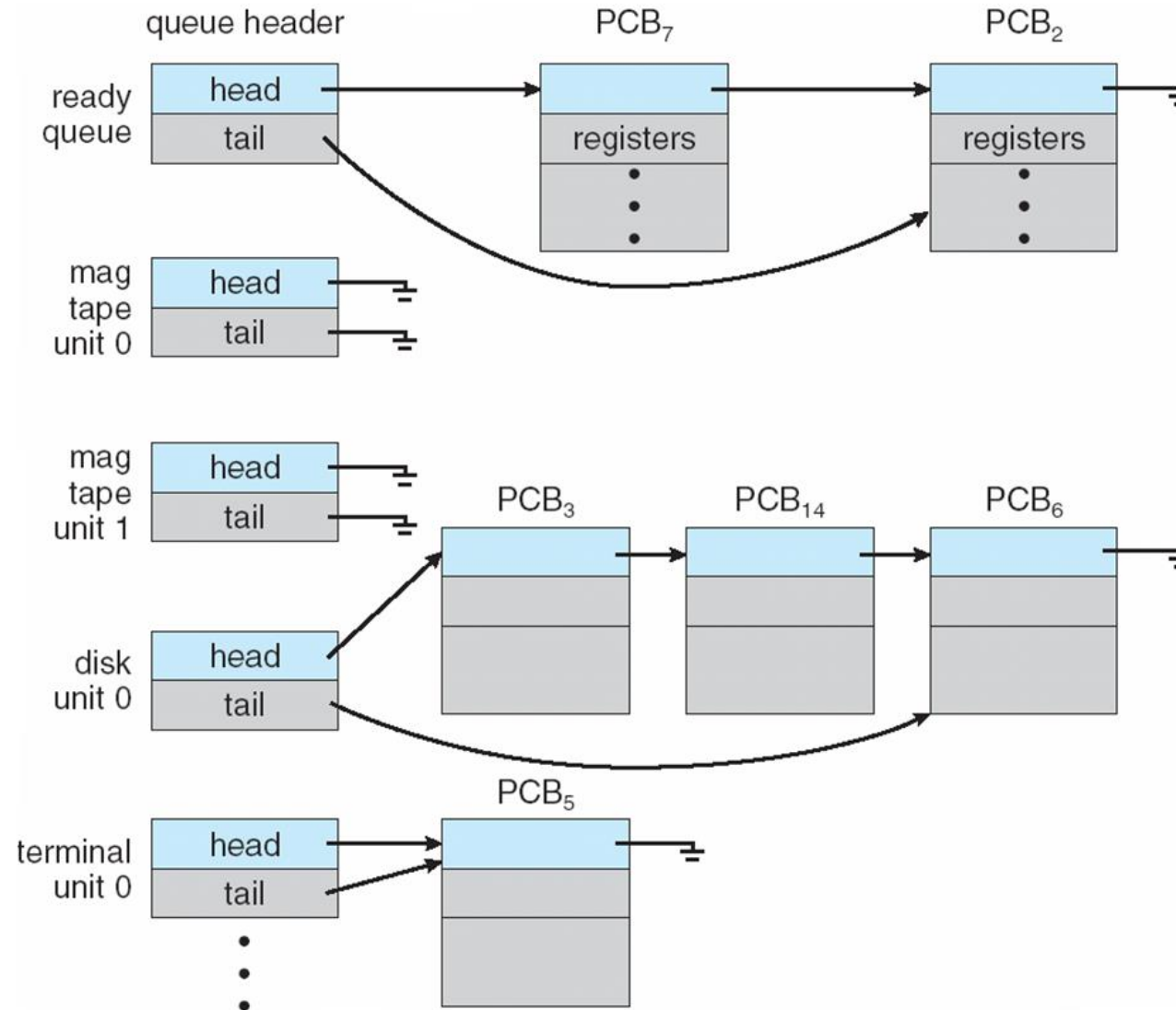
- Our goals are to:
 - Maximize CPU use
 - Give processes the CPU time that they need
- **Process scheduler** selects among available processes for next execution on CPU

Process Scheduler

Maintains **scheduling queues** of processes

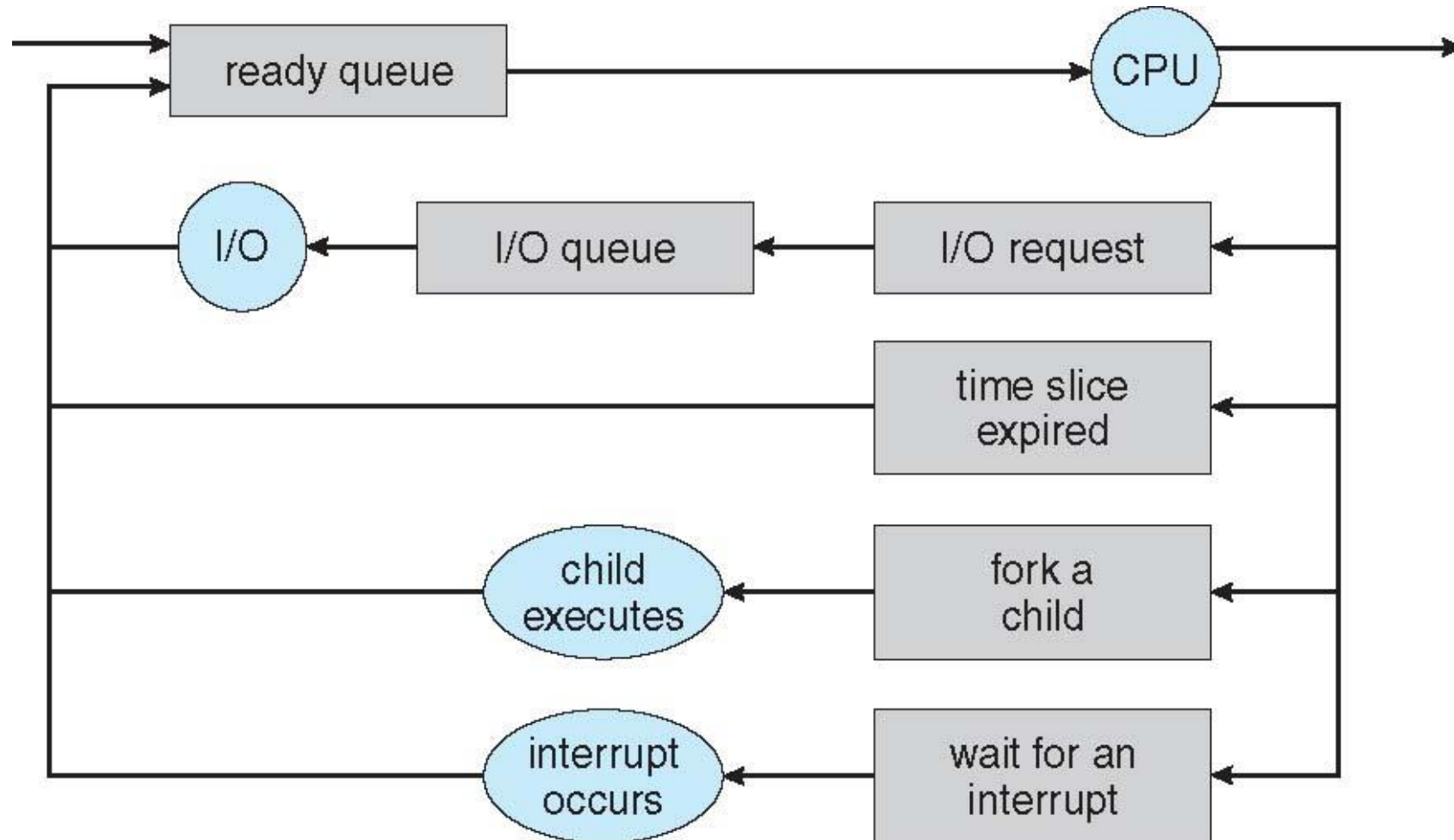
- **Job queue:** set of all processes in the system
- **Ready queue:** set of all processes residing in main memory, ready and waiting to execute
- **Device queues:** set of processes waiting for an I/O device
- Processes migrate among the various queues, depending (in part) on their state

Ready Queue And Various I/O Device Queues



Process Scheduling

Queueing diagram represents queues, resources, flows



Scheduler Components

Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU

- Sometimes the only scheduler in a system
- Short-term scheduler is invoked frequently (milliseconds), so it must be fast

Scheduler Components

Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue

- Long-term scheduler is invoked infrequently (seconds, minutes), so it may respond slowly
- The long-term scheduler controls the degree of multiprogramming
- Most important in (older) resource-bound systems

Scheduler Components

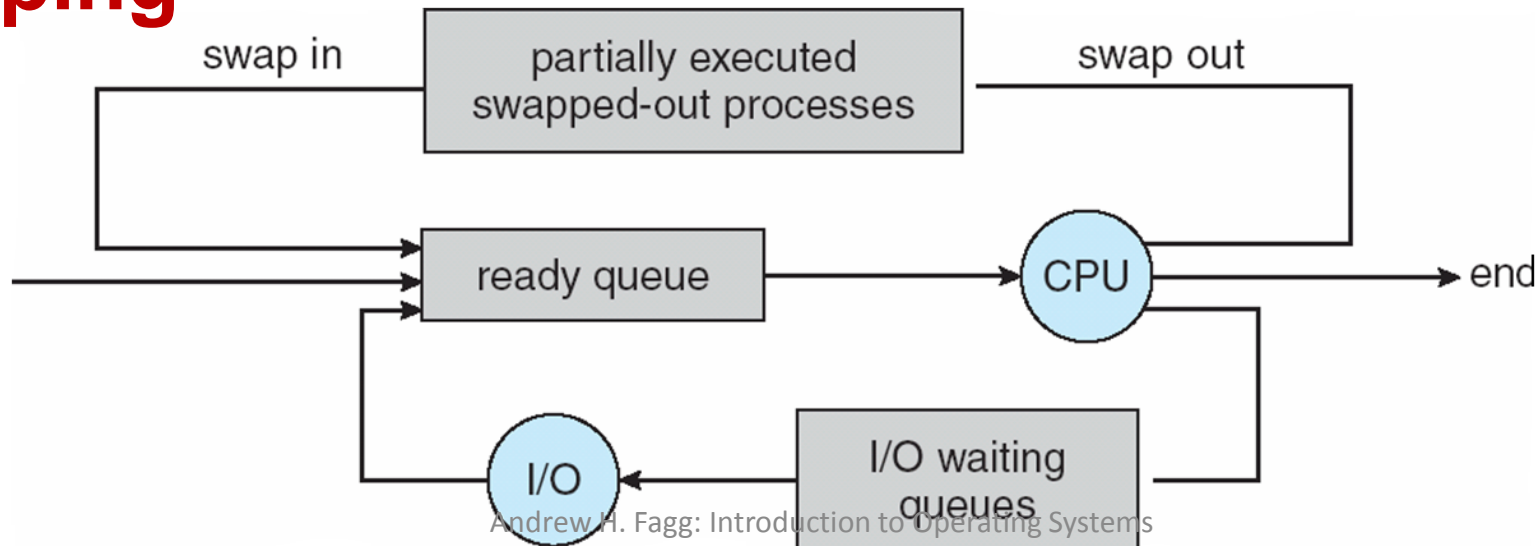
Processes can be described as either:

- **I/O-bound**: spends more time doing I/O than computations, many short CPU bursts
- **CPU-bound**: spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good process mix
 - Goal: keep both I/O and CPU resources as busy as possible

Medium Term Scheduling

Medium-term scheduler can be added if degree of multiple programming needs to decrease

- Remove process temporarily from memory, store on disk, bring back in from disk to continue execution:
swapping



Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run; others are suspended
- Due to screen real estate and user interface limits, iOS provides for:
 - Single foreground process: controlled via user interface
 - Multiple background processes: in memory, running, but not on the display, and with limits
 - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background processes, with fewer limits

Context Switching

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
 - The more complex the OS and the PCB, the longer the context switch
 - Time is dependent on hardware support
 - Some hardware provides multiple sets of registers per CPU
 - Allows multiple contexts to be loaded at once

Operations on Processes

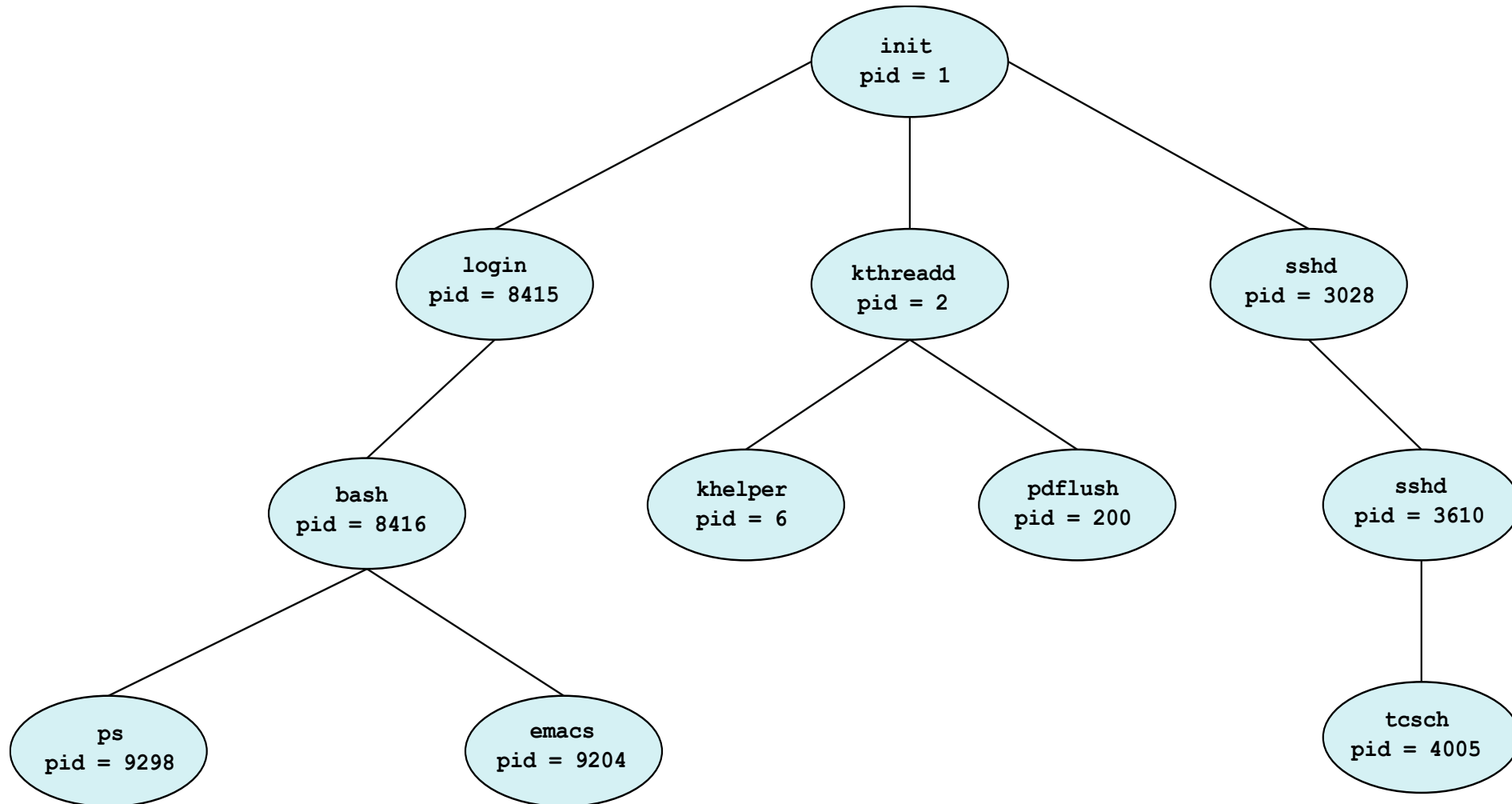
Process Creation

- **Parent** process creates **child** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**

Process Creation

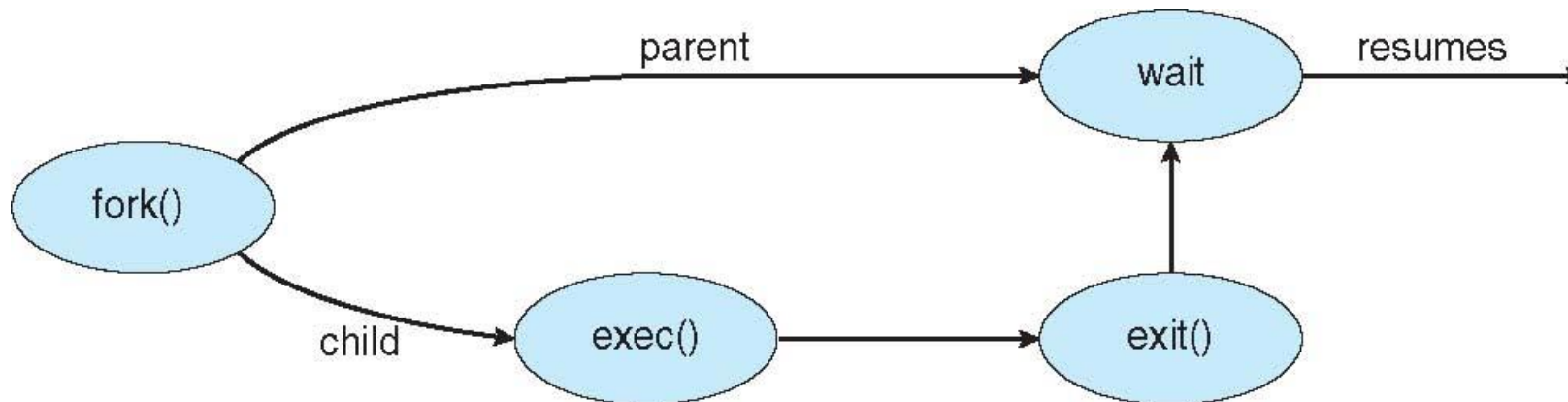
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution options
 - Parent and children execute concurrently
 - Parent waits until children terminate

Process Tree



Process Creation (Cont.)

- Address space
 - Child duplicate of parent
 - Child then has a program loaded into it
- UNIX examples
 - `fork()` system call creates new process
 - `exec()` system call used after a `fork()` to replace the process' memory space with a new program

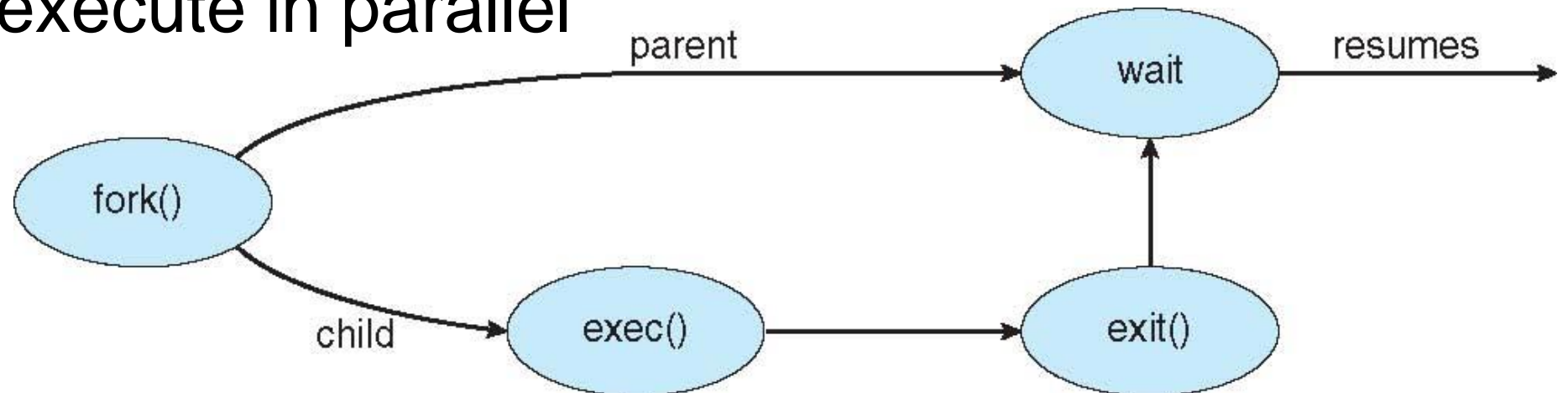


fork live demo

fork()

fork() creates an identical process to the one that called fork()

- Same program
- Same state, including open file descriptors
- Exception: fork() returns 0 to the child; and a positive integer to the parent
- Processes execute in parallel



C Program Forking a Child Process

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

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

Live execlp() demo

execvp

```
execvp(char *command, char *argv0, char *argv1, ... NULL)
```

- Replaces the currently executing program with a new program & begins execution
- command is a string that references an executable file
 - Can be absolute or relative path
 - Relative path: use the **path** environment variable to find the executable
- argv0, argv1, ... are arguments to be passed to the executable
 - Don't forget the NULL at the end!

execlp

`execlp(char *command, char *argv0, char *argv1, ... NULL)`

- If this function is successful in finding the specified executable, it does not return!

system()

`system(char *command)`

- Command specifies the executable and arguments in one string
- Relative path: looks up the executable in the path environment variable
- First calls `fork()`, then `execvp()`
- Always returns
 - Return value is the exit code from the specified command

Live system() demonstration

Process Termination

Process executes last statement and then asks the operating system to delete it using the `exit()` system call.

- `exit(-1)`: -1 is the exit code returned by the process
- Status data from child can be passed to the parent
 - `pid = wait(&status) ;`
 - status contains information about the reason for termination
- Process' resources are deallocated by operating system

Process Termination

Some operating systems do not allow child processes to exist if its parent has terminated (including Linux). If a process terminates, then all its children must also be terminated.

- If parent is executing, but not waiting (did not invoke `wait()`) and the child process ends, then the child process is a **zombie**
- If parent terminated without invoking `wait()`, the child process is an **orphan**
 - Orphans become children of the **init** process

Cooperating Processes

- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience

Multiprocess Architecture – Chrome Browser

Many web browsers ran as single process (some still do)

- If one web site causes trouble, the entire browser can hang or crash

Multiprocess Architecture – Chrome Browser

Google Chrome Browser is multiprocess with 3 different types of processes:

- **Browser** process manages user interface, disk and network I/O
- **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
 - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
- **Plug-in** process for each type of plug-in

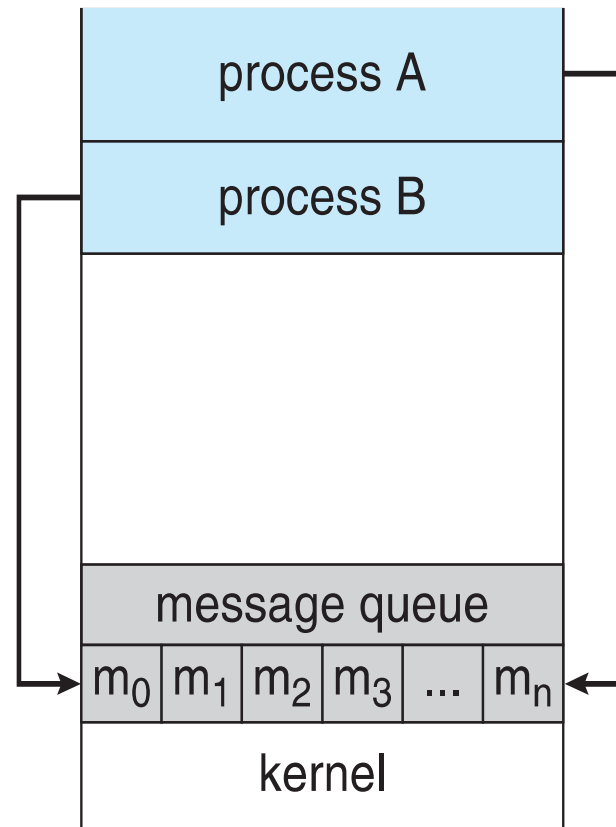


Interprocess Communication

- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
 - **Shared memory**
 - **Message passing**

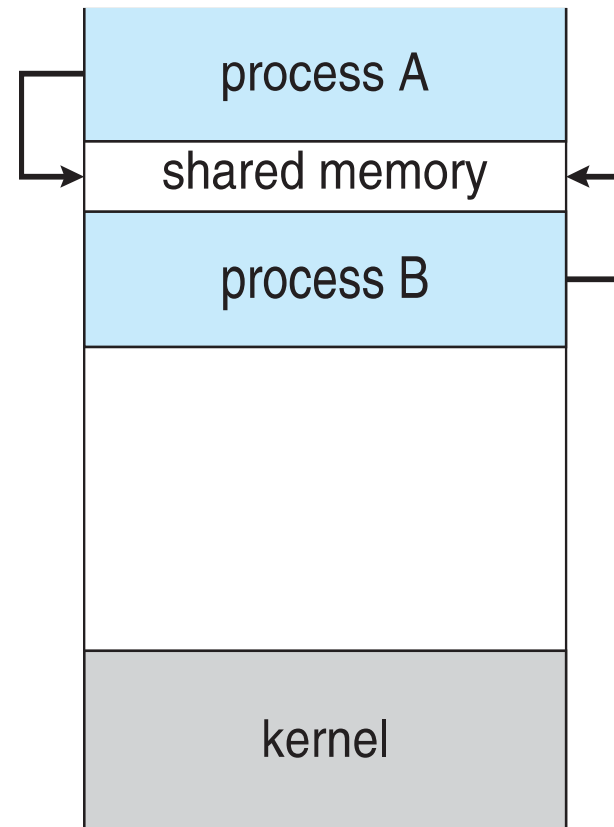
Communication Models

Message Passing



(a)

Shared Memory



(b)

Producer-Consumer Problem

- Producer: process generates data through some mechanism
- Consumer: process uses data generated by another

Producer-Consumer Problem

Typical approach: implement a data buffer from the producer to the consumer

- **unbounded-buffer** places no practical limit on the size of the buffer
- **bounded-buffer** assumes that there is a fixed buffer size

Circular/Shared Buffer of Items

- Items are instances of type **item**

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- in = the next location to place a new item
- out = the next location to remove an item from
- Both the producer and consumer processes have access to this buffer

Circular/Shared Buffer of Items

- $\text{in} == \text{out}$: no items in the buffer
- $(\text{in} + 1) \% \text{BUFFER_SIZE} = \text{out}$: buffer is full

Circular Buffer: Producer

```
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced; // Copy item
    in = (in + 1) % BUFFER_SIZE;
}
```

Circular Buffer: Consumer

```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

Shared Memory

- Shared memory: an area of memory shared among the processes that wish to communicate
- The communication is entirely under the control of the users processes not the operating system
 - Good for efficiency
 - Lots of opportunities for bugs
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access the shared memory

Interprocess Communication: Message Passing

Mechanism for processes to communicate and to synchronize their actions

- Message system: processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - send(message)
 - receive(message)
- The message size is either fixed or variable

Message Passing

- If processes P and Q wish to communicate, they need to:
 - Establish a ***communication link*** between them
 - Exchange messages via send/receive
- Implementation issues:
 - How are links established?
 - Can a link be associated with more than two processes?
 - How many links can there be between every pair of communicating processes?
 - What is the capacity of a link? (buffer)
 - Is the size of a message that the link can accommodate fixed or variable?
 - Is a link unidirectional or bi-directional?

Message Passing

- Implementation of communication link
 - Physical:
 - Shared memory
 - Hardware bus
 - Network
 - Logical:
 - Direct or indirect
 - Synchronous or asynchronous
 - Automatic or explicit buffering

Direct Communication Model

- Processes must name each other explicitly:
 - **send** (*P*, *message*): send a message to process *P*
 - **receive**(*Q*, *message*): receive a message from process *Q*
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional

Indirect Communication Model

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
 - create a new mailbox (port)
 - send and receive messages through mailbox
 - destroy a mailbox
- Primitives are defined as:
 - send**(*A, message*) – send a message to mailbox *A*
 - receive**(*A, message*) – receive a message from mailbox *A*

Mailbox Sharing

- *Scenario*
 - P_1 , P_2 , and P_3 share mailbox A
 - P_1 sends; P_2 and P_3 receive
 - Who gets the message?
- *Solutions*
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
 - Blocking send: the sender is blocked until the message is received
 - Blocking receive: the receiver is blocked until a message is available

Synchronization

- Non-blocking is considered asynchronous
 - Non-blocking send: the sender sends the message and continues
 - Non-blocking receive: the receiver receives:
 - A valid message, or
 - Null message
- Different combinations possible
 - If both send and receive are blocking, we have a **rendezvous**

Synchronization with Rendezvous

Producer-consumer becomes trivial

```
message next_produced;
while (true) {
    /* produce an item in next produced */
    send(next_produced);
}
```

```
message next_consumed;
while (true) {
    receive(next_consumed);

    /* consume the item in next consumed */
}
```

Buffering

- Queue of messages attached to the link
- Implemented in one of three ways:
 - Zero capacity: no messages are queued on a link.
Sender must wait for receiver (rendezvous)
 - Bounded capacity: finite length of n messages
Sender must wait if link full
 - Unbounded capacity: infinite length
Sender never waits

Shared Memory in POSIX (includes Linux)

- Process first creates shared memory segment (or opens an existing one):

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- Set the size of the object:

```
ftruncate(shm_fd, 4096);
```

- Create a pointer to the shared memory:

```
mmap(shm_fd, 0, 4096 PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

- Now the process can write to the shared memory

```
sprintf(shared memory, "Writing to shared memory");
```

Shared Memory Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);

    return 0;
}
```

Shared Memory Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Client-Server Model of Communication

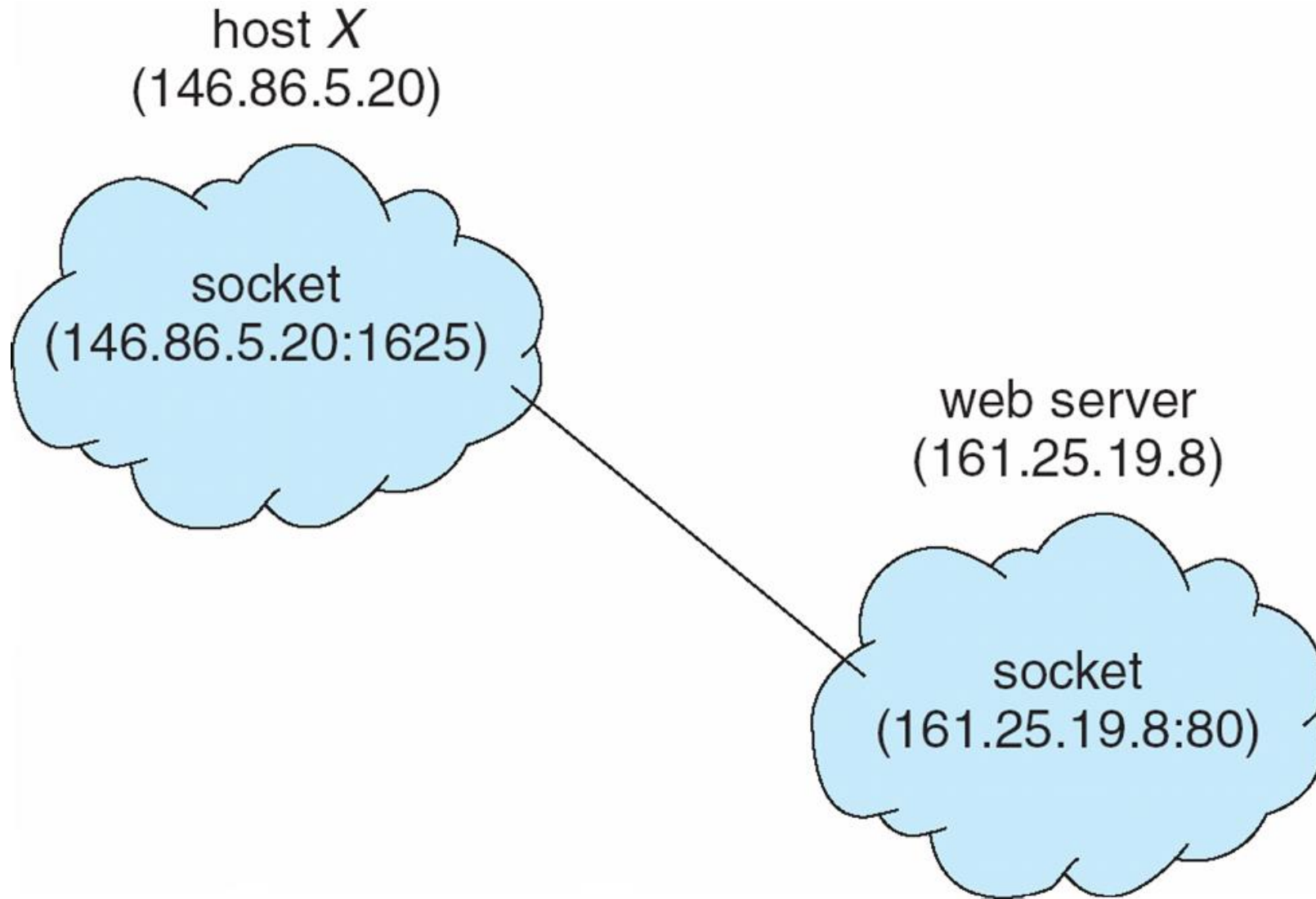
- Built on top of Producer/Consumer
- Server: provides some service (data, computation)
- Client: requests actions on the part of the server
- Implementation choices include:
 - Sockets
 - Pipes
 - Remote Procedure Calls
 - Remote Method Invocation (Java)

Sockets

A socket is defined as an endpoint for communication

- Identified by a concatenation of IP address and port: a number included at start of message packet to differentiate network services on a host
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets
- All ports below 1024 are well known and are used for standard services
- Special IP address 127.0.0.1 (loopback) to refer to system on which process is running

Socket Communication



Remote Procedure Calls

- From the programmer's perspective, they appear as functions/methods that take arguments and return a value
- Under the hood, this function call:
 - Contacts a server
 - Sends the arguments to the server
 - Server does the work and sends the result back
 - Return the return value back to the client

Pipes

- Act as a conduit allowing two processes to communicate
- Issues:
 - Is communication unidirectional or bidirectional?
 - In the case of two-way communication, is it half or full-duplex?
 - Must there exist a relationship (i.e., parent-child) between the communicating processes?
 - Can the pipes be used over a network?

Pipes

- Ordinary pipes: cannot be accessed from outside the process that created it.
 - Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes: can be accessed without a parent-child relationship.

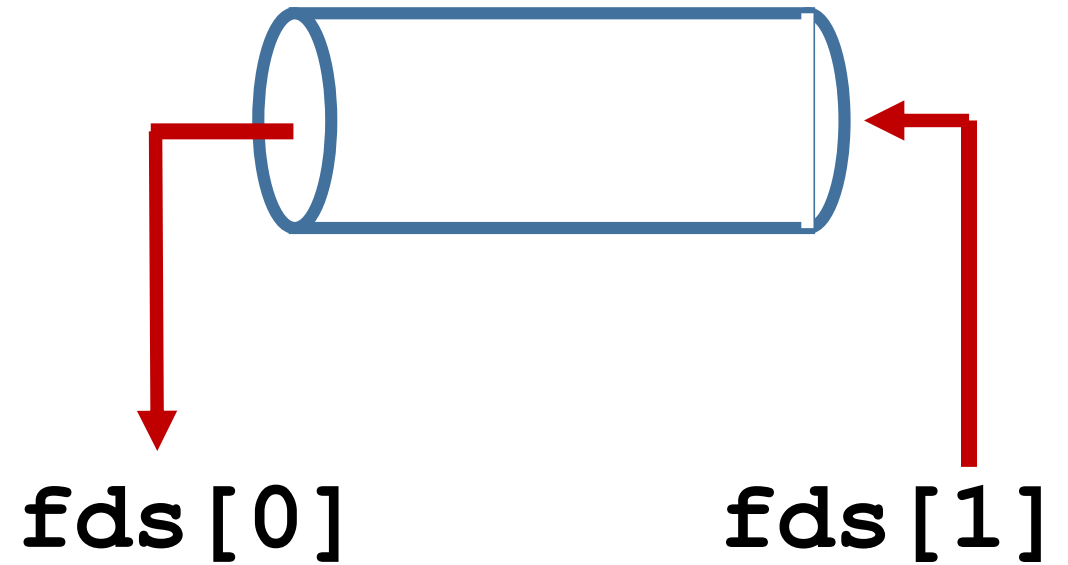
Ordinary Pipes

Ordinary Pipes allow communication in standard producer-consumer style

- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional

Ordinary Pipe

```
int fds[2];  
int ret = pipe(fds);  
if(ret < 0) exit(-1);  
// Now use the pipe
```



- `fds[0]`: output from pipe
- `fds[1]`: input to pipe

Ordinary Pipes for Communication

- The pipe is implemented inside the kernel (so, it does not exist within the process)
- However, the process maintains this pair of file descriptors, which allow it to reference the pipe

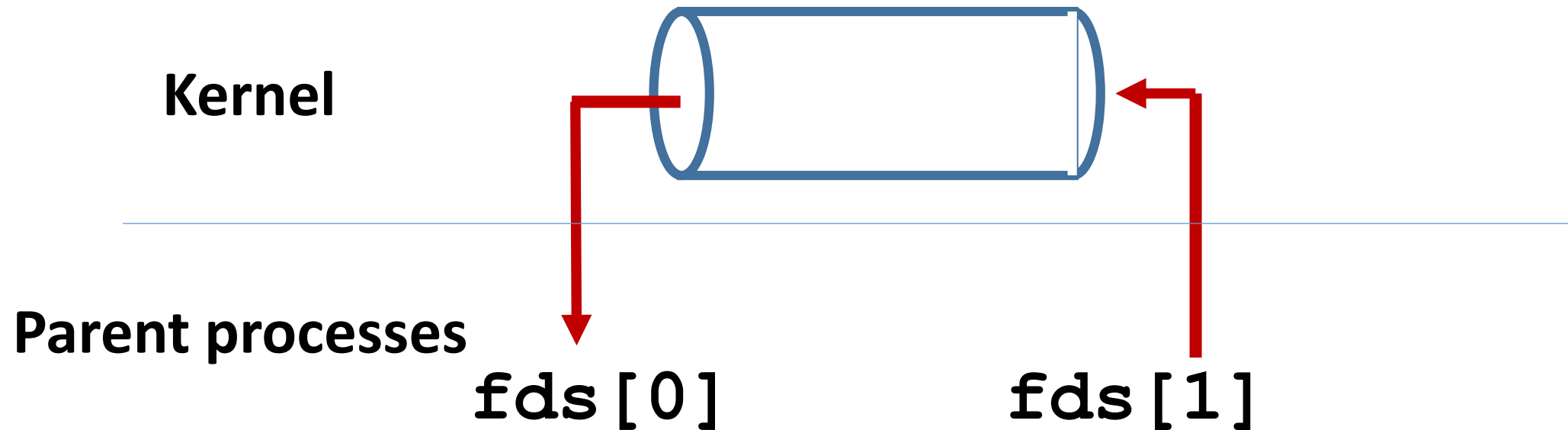
Ordinary Pipes for Communication

- The file descriptors cannot be shared outside of the process
- But: if the process forks(), then both the parent and child will have copies of the file descriptors!
 - And these reference the same pipe

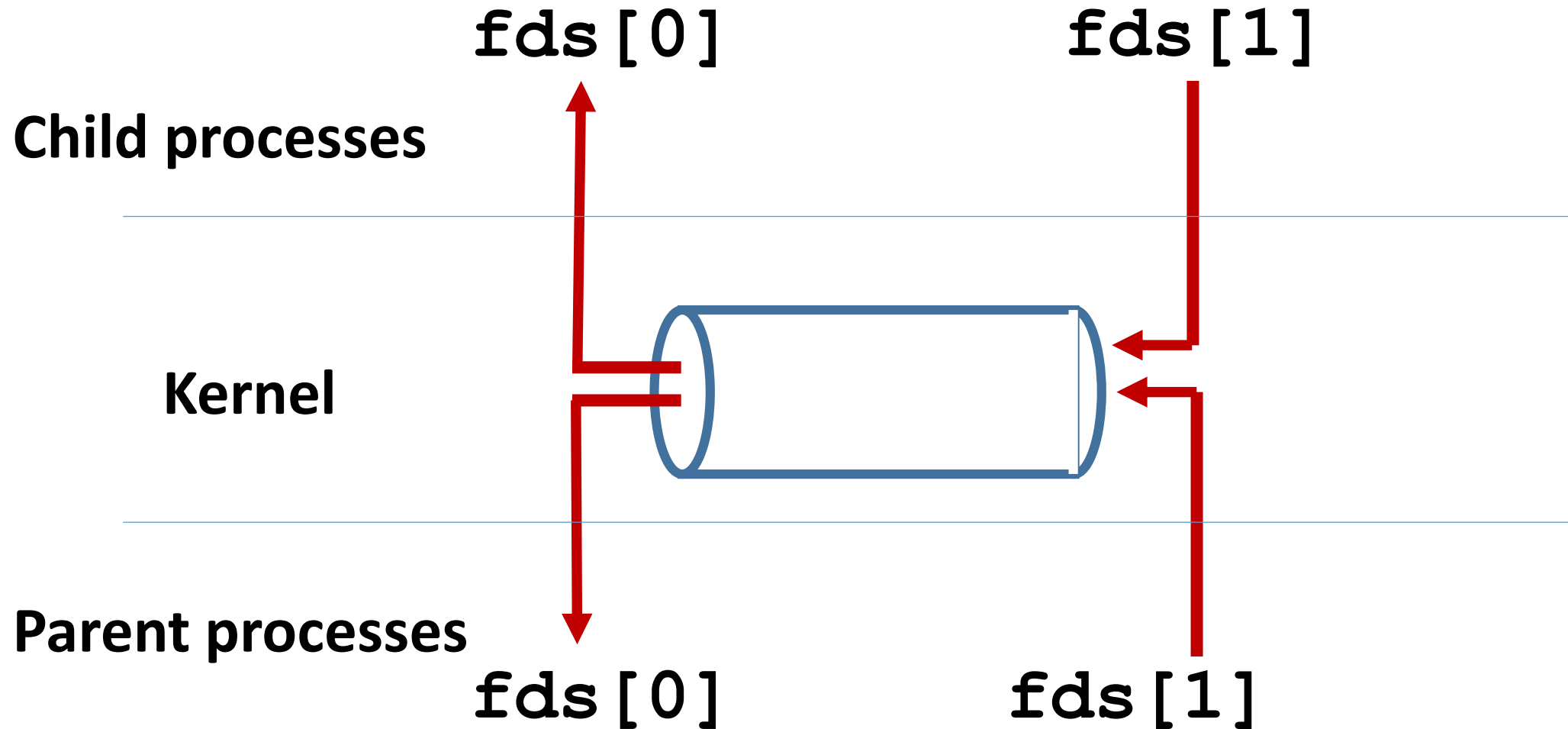
Ordinary Pipe with Fork()

```
int fds[2];
int ret = pipe(fds);
if(ret < 0) exit(-1);
// Now use the pipe
int pid = fork()
if(pid > 0) {           // Note: leaving off error case
    // parent code
}else {
    // child code
}
```


Before Fork



After Fork

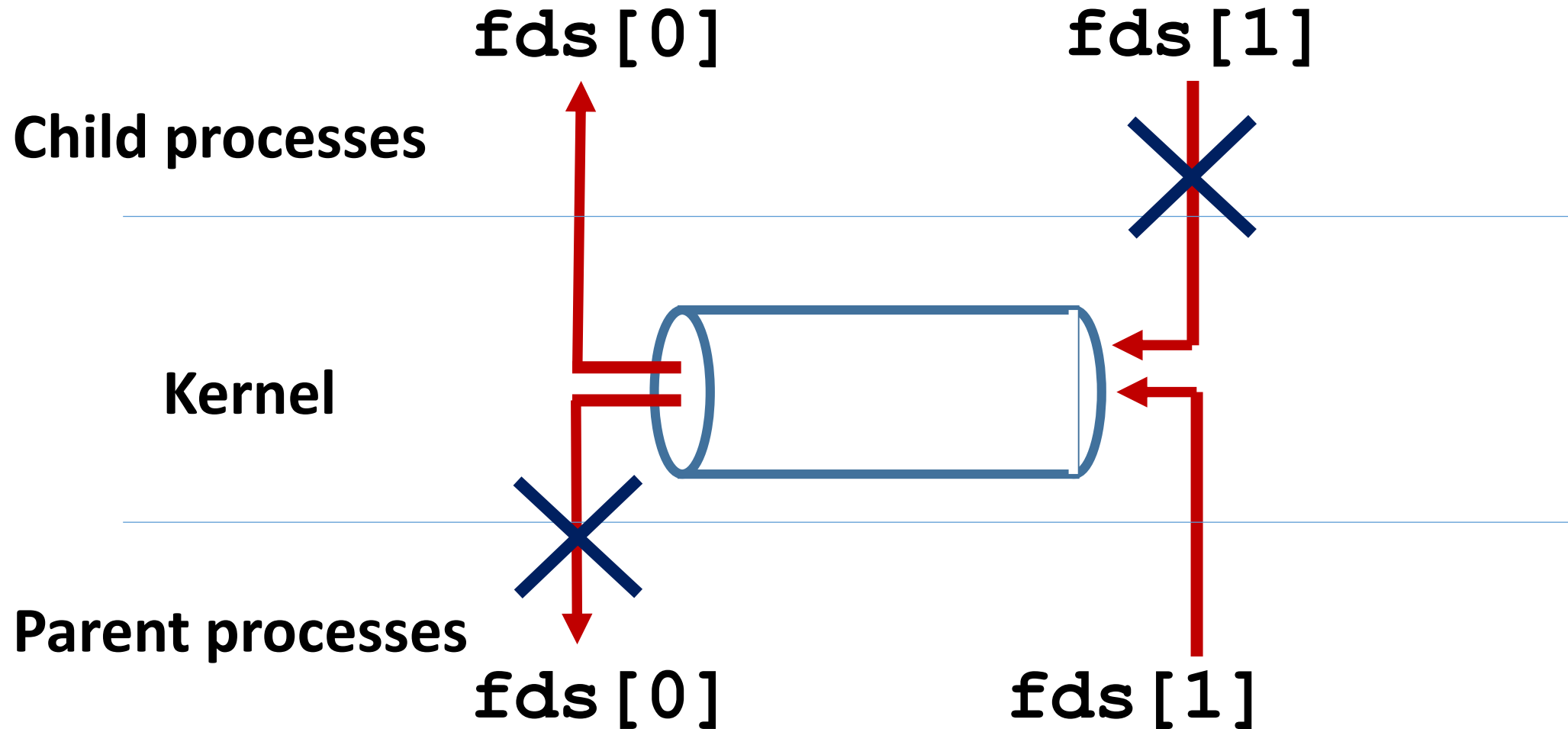


Pipes

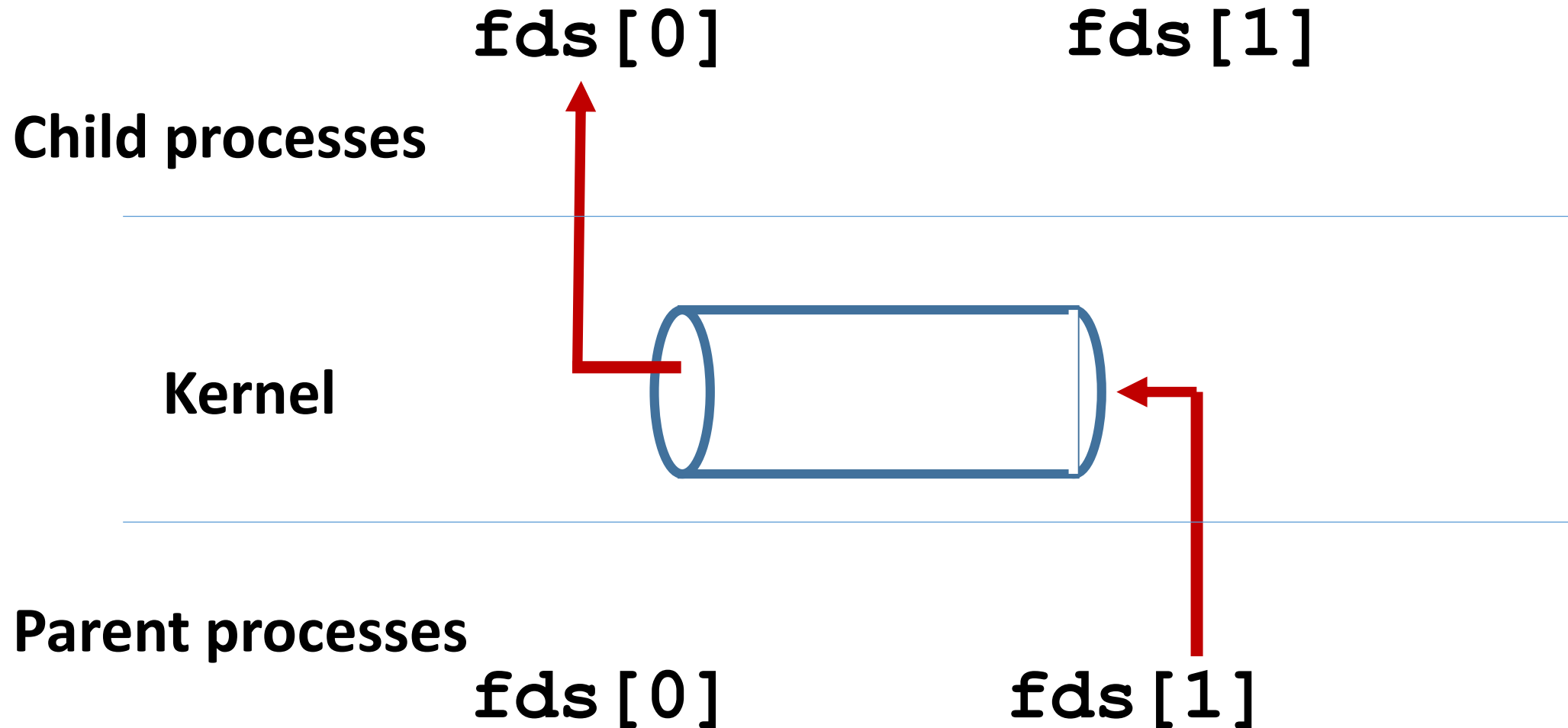
For our purposes, we will assume:

- Pipes will only work properly with one reader and one writer (this is the typical use case)
- This means, after the fork, each of the parent and child will close one of the two file descriptors

After Fork: Parent is the Producer



After Fork: Parent is the Producer



After Fork: Parent is the Producer

```
// Now use the pipe
int pid = fork()
if(pid > 0) {                                // Note: leaving off error case
    // parent code
    close(fds[0])
    :
}else {
    // child code
    close(fds[1])
    :
}
```

After Fork: Parent is the Producer

After fork and closing:

- Parent can write bytes to `fds[1]`
- Child can read these bytes from `fds[0]`

After Fork: Parent is the Producer

- The pipe has a buffer: it will hold written bytes until they are read
- If the writer closes the pipe, then
 - The reader gets to read the remaining bytes
 - But then will see an EOF
- Windows calls ordinary pipes ***anonymous pipes***

Pipe demos

Named Pipes

Named Pipes are more powerful than ordinary pipes

- Communication **can** be bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Named Pipes in Unix

- Access points exist in the file system
 - Open them just as you would a file!
 - Use read()/write() to receive/send data
- Can have multiple readers/writers
 - A message is delivered to one randomly selected reader
 - So, effectively, they are bidirectional
 - **However, we will use them as unidirectional pipes**
- Create at the command line (or programmatically):
`mkfifo [NAME]`

- Quick named pipe demo