

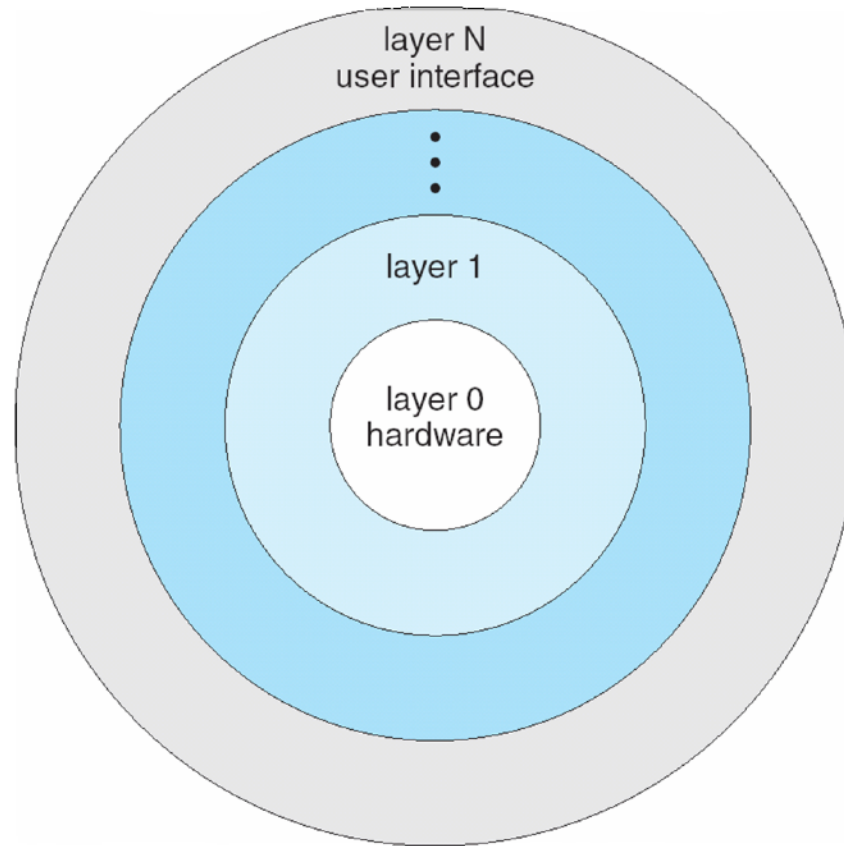
Processes and More

CSCI 315 Operating Systems Design
Department of Computer Science

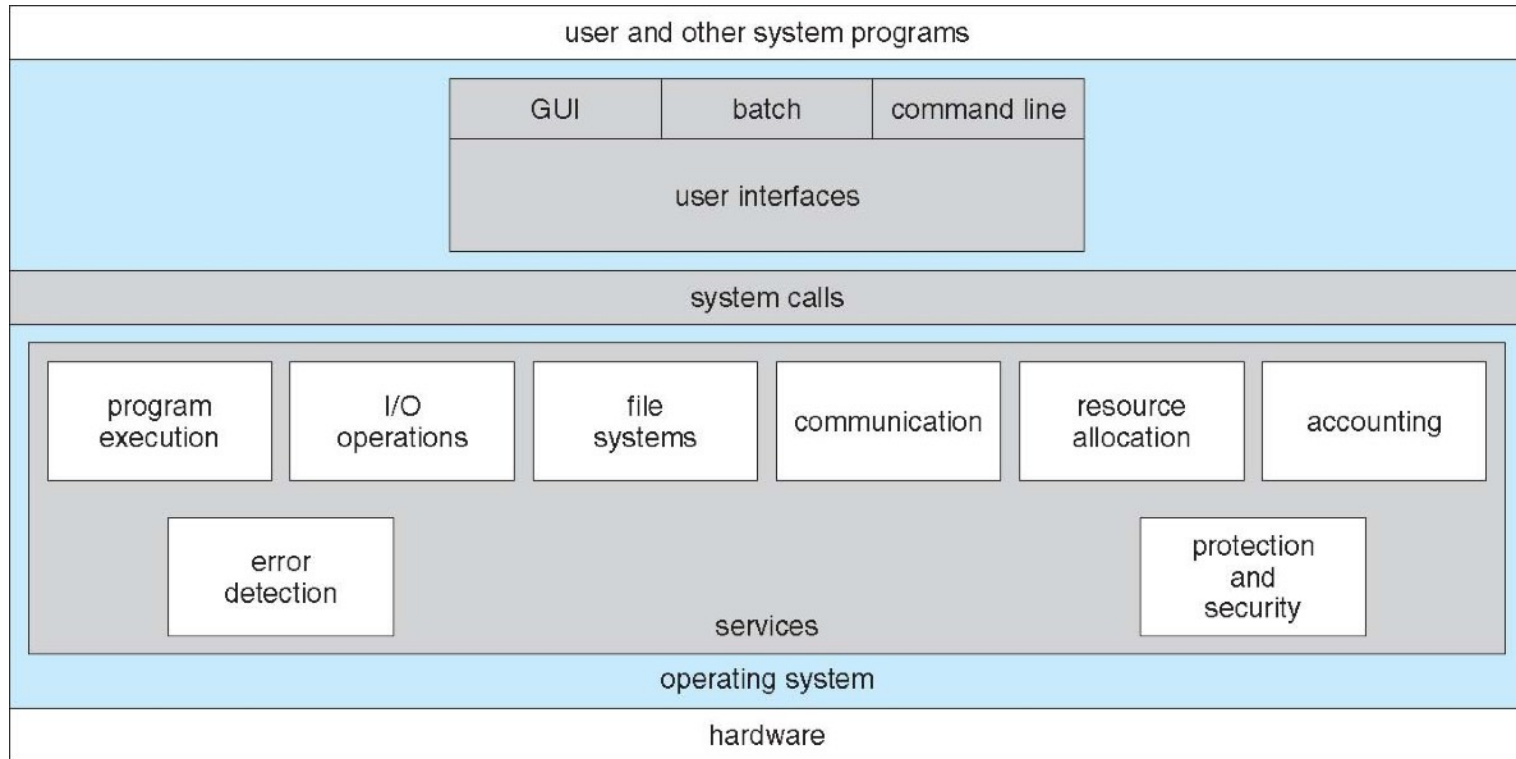
Notice: The slides for this lecture have been largely based on those accompanying the textbook *Operating Systems Concepts*, 10th ed., by Silberschatz, Galvin, and Gagne. Many, if not all, the illustrations contained in this presentation come from this source.



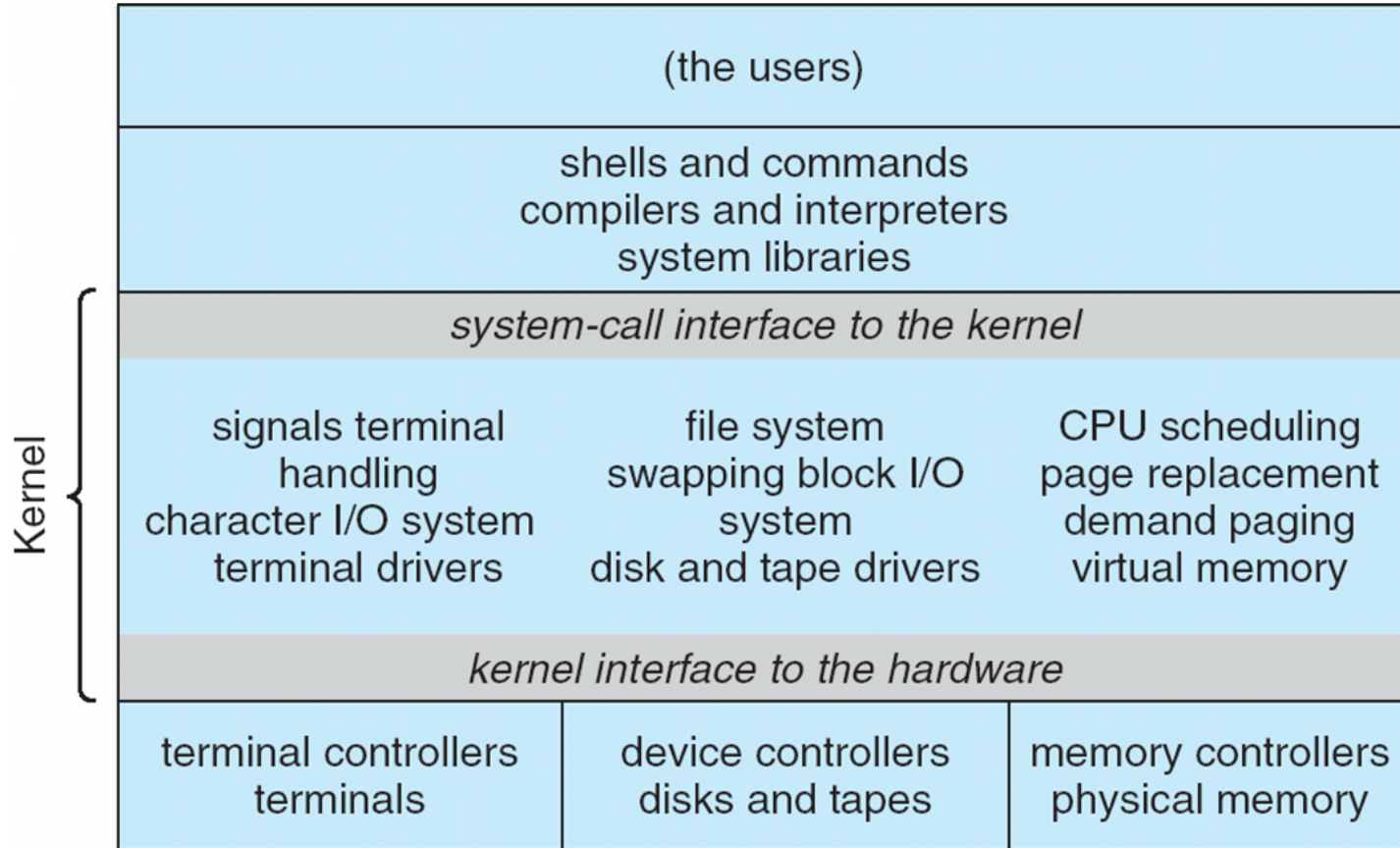
Abstractions and Layers



OS Services



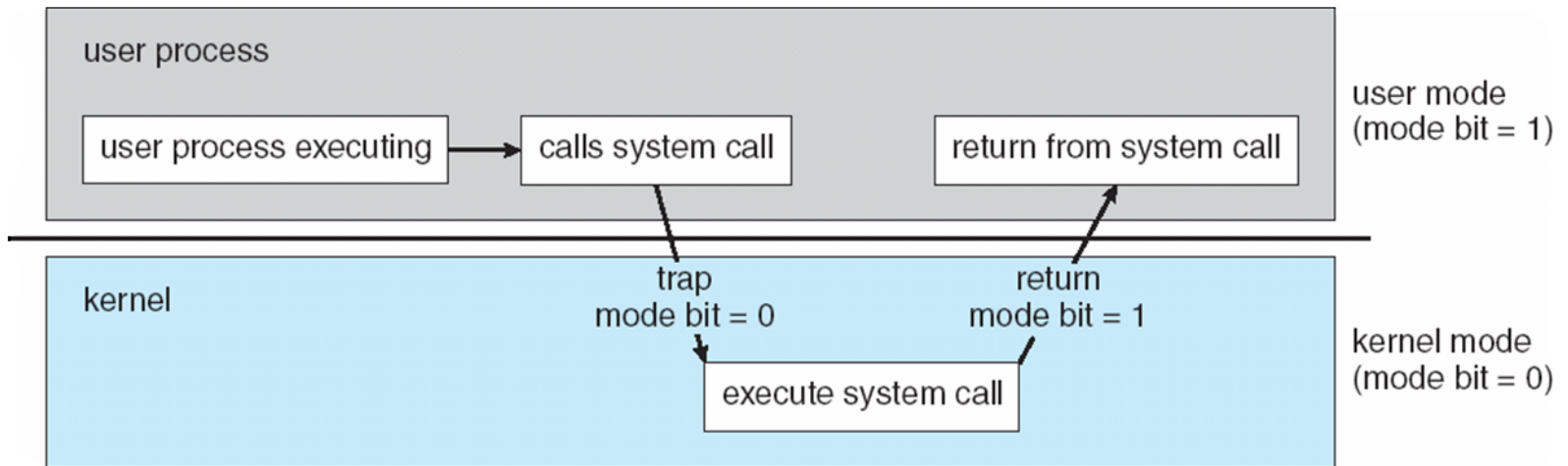
Unix Structure



OS Operations

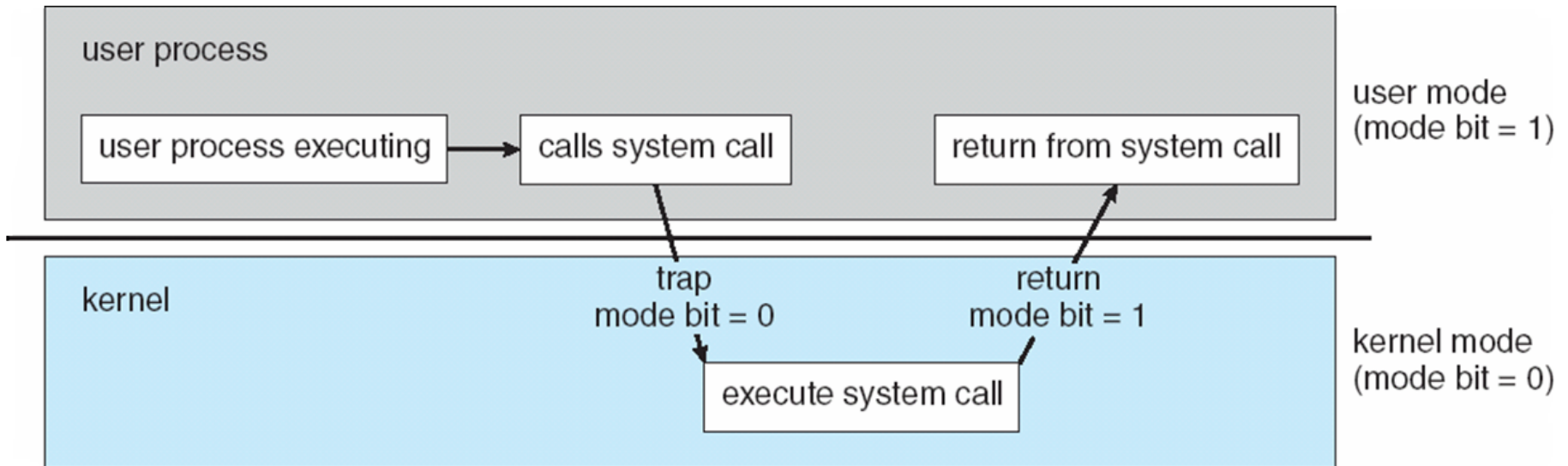
- **Interrupt driven** by hardware
- Software error or request creates **exception** or **trap**
 - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
 - **User mode** and **kernel mode**
 - **Mode bit** provided by hardware
 - Provides ability to distinguish when system is running user code or kernel code
 - Some instructions designated as **privileged**, only executable in kernel mode
 - System call changes mode to kernel, return from call resets it to user
- Increasingly CPUs support multi-mode operations
 - i.e. **virtual machine manager (VMM)** mode for guest **VMs**

User and Kernel Modes

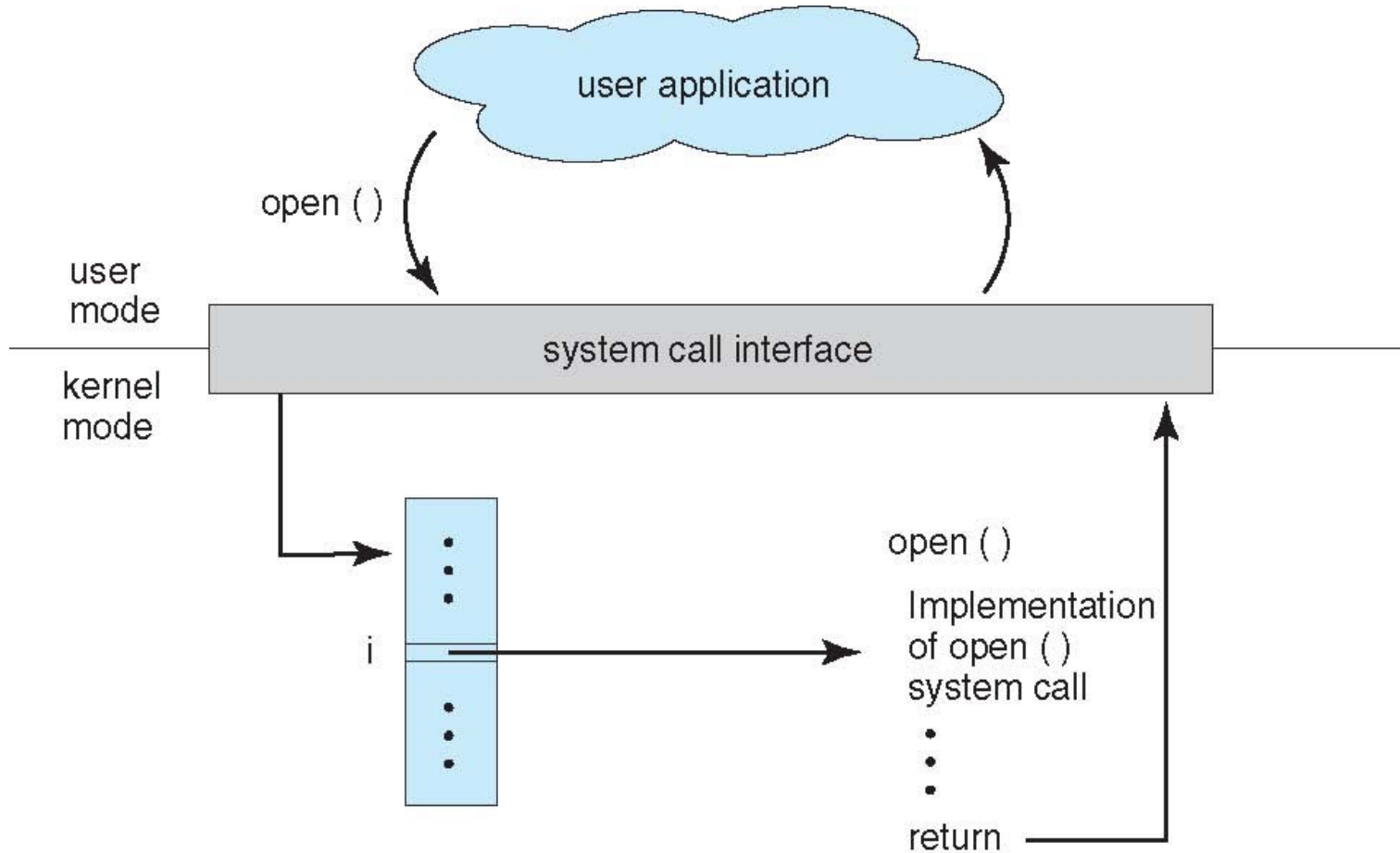


Hardware Support for the OS

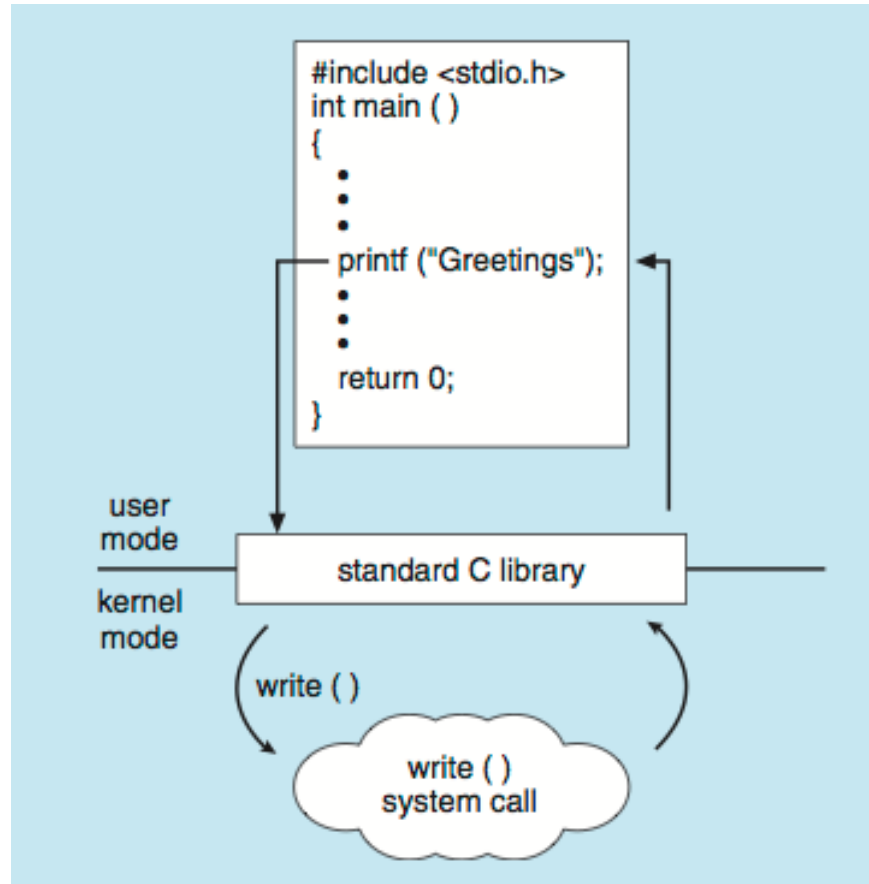
- **Two classes of instructions:** one class for anyone to use, others with **privileged** use (for the OS kernel).
- Need to be able to switch between **user mode** and **kernel mode**.
- If a user runs a privileged instruction, an exception is raised.
- To switch to kernel mode, you need to trap to the kernel.



System Calls and the OS



System Calls and Libraries



strace

```
1. perrone@linuxremote1:~ (ssh)
STRACE(1) STRACE(1)

NAME
    strace - trace system calls and signals

SYNOPSIS
    strace [ -dDffhiqrtttTvVxx ] [ -acolumn ] [ -eexpr ] ... [ -ofile ] [
    -ppid ] ... [ -sstrsize ] [ -uusername ] [ -Evar=val ] ... [ -Evar ]
    ... [ command [ arg ... ] ]

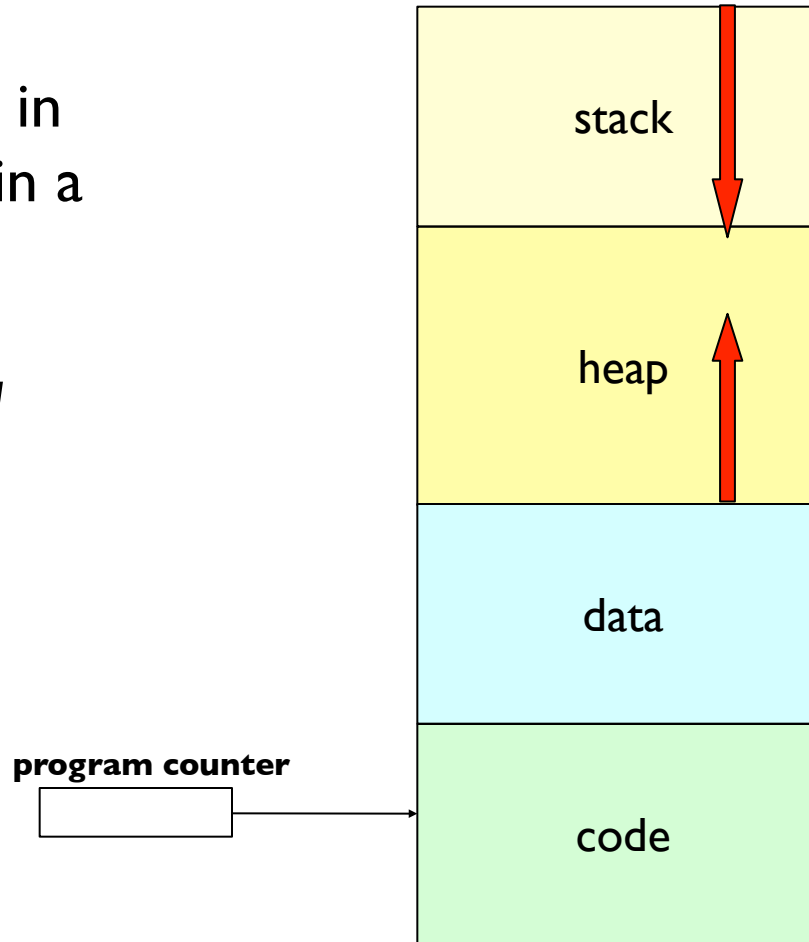
    strace -c [ -D ] [ -eexpr ] ... [ -Ooverhead ] [ -Ssortby ] [ command
    [ arg ... ] ]

DESCRIPTION
    In the simplest case strace runs the specified command until it exits.
    It intercepts and records the system calls which are called by a process
    and the signals which are received by a process. The name of each system
    call, its arguments and its return value are printed on standard error or
    to the file specified with the -o option.

    strace is a useful diagnostic, instructional, and debugging tool. System
    administrators, diagnosticians and trouble-shooters will find it invaluable
    for solving problems with programs for which the source is not readily
    available since they do not need to be recompiled in order
```

Process Concept

- Process – a program in execution; the code in a process executes sequentially. (*Ahem, mostly. To be discussed later.*)
- A process includes:
 - program counter,
 - stack,
 - data section.



Creating *processes* in Unix

(Always RTFMP)

```
perrone@linuxremote1:~ (ssh)
Linux Programmer's Manual
FORK(2)
NAME
    fork - create a child process
SYNOPSIS
    #include <unistd.h>

    pid_t fork(void);
DESCRIPTION
    fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent, except for the following points:

    * The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)).

    * The child's parent process ID is the same as the parent's process ID.

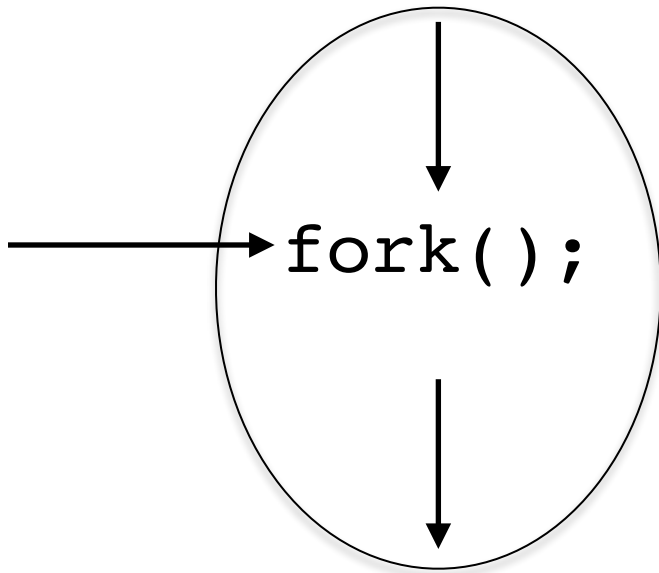
    * The child does not inherit its parent's memory locks (mlock(2), mlockall(2)).

    * Process resource utilizations (getrusage(2)) and CPU time counters (times(2)) are reset to zero in the child.

    * The child's set of pending signals is initially empty (sigpending(2)).
Manual page fork(2) line 1 (press h for help or q to quit)
```

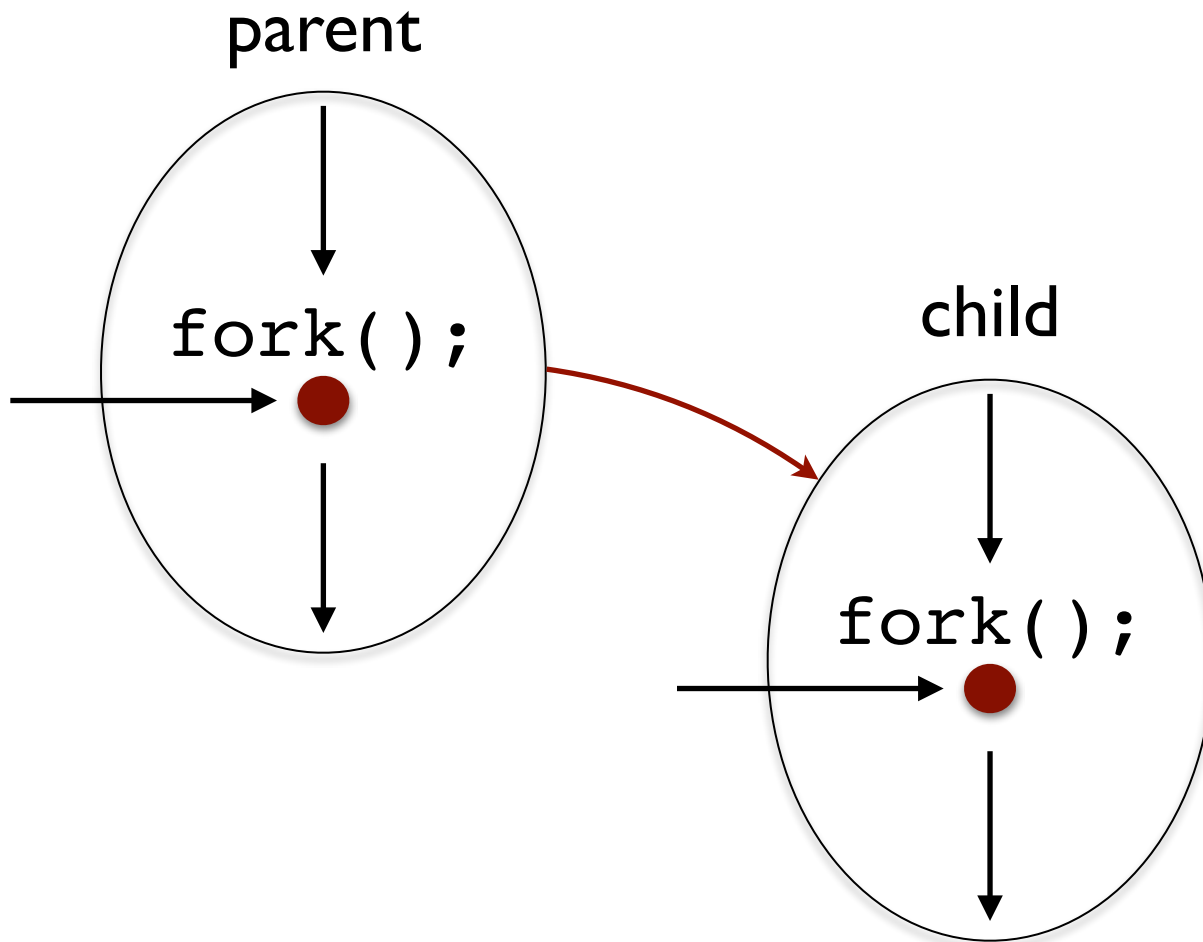
Forking

(yeah, it's a thing, a Unix thing)



Forking

(yeah, it's a thing, a Unix thing)



Forking

(what that return value is for)

parent

```
int pid;
```



```
pid = fork();
```



Forking

(what that return value is for)

parent

```
int pid;
```

```
pid = fork();
```

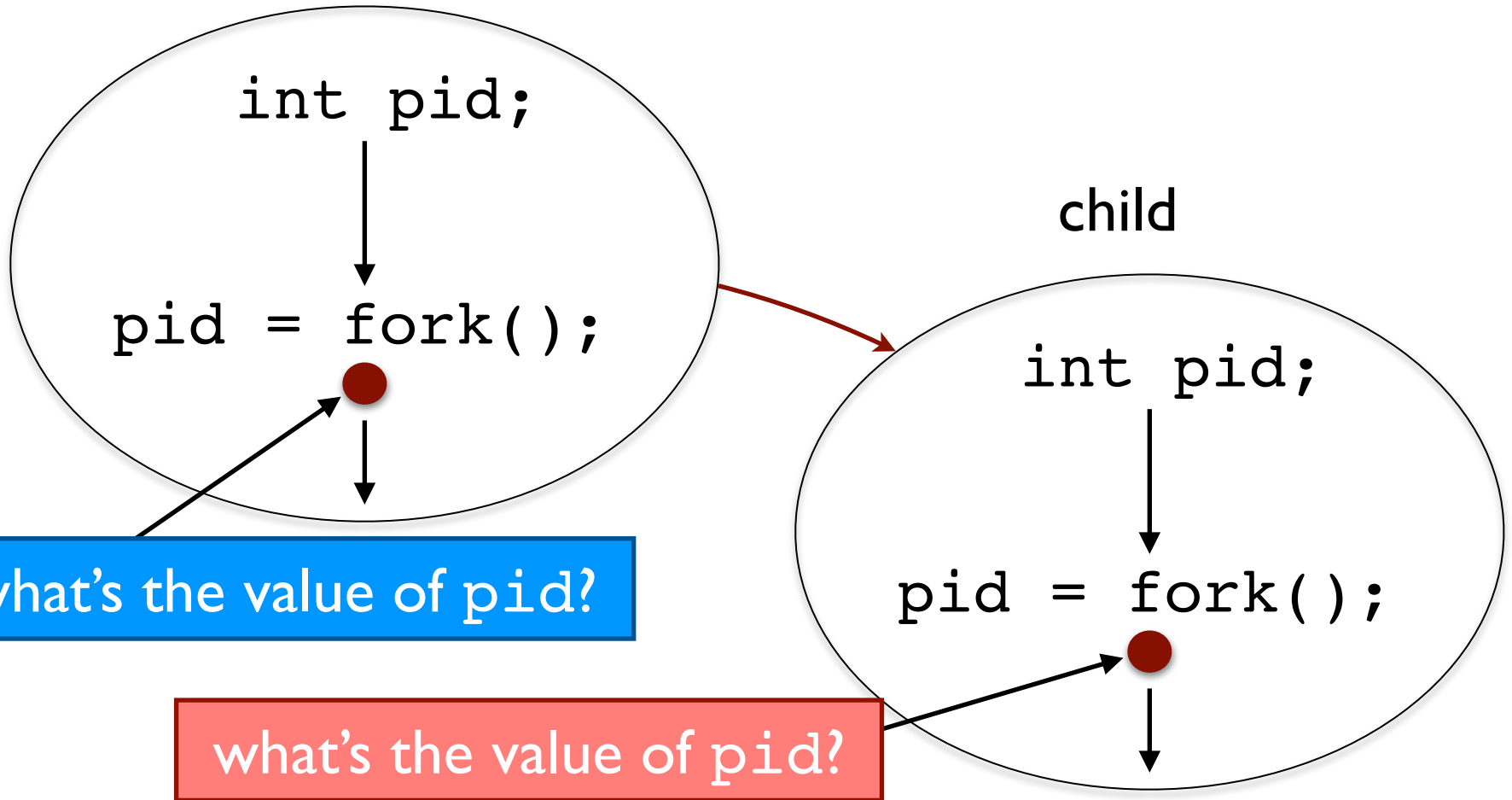
what's the value of pid?

child

```
int pid;
```

```
pid = fork();
```

what's the value of pid?



Using fork safely

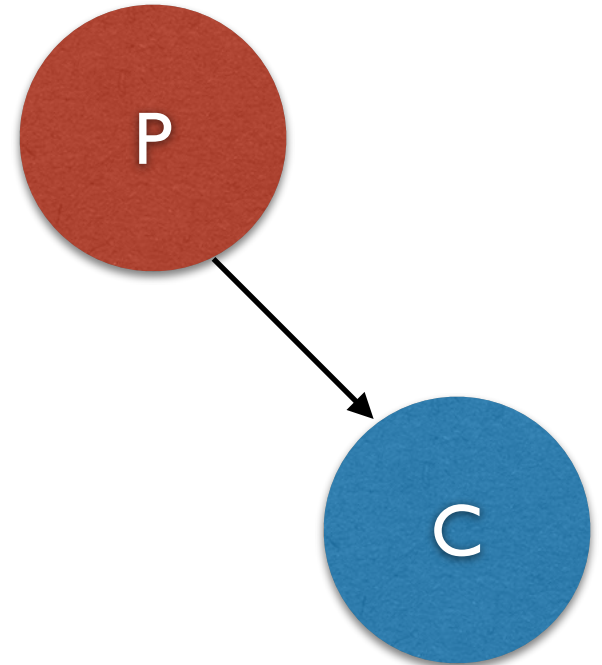
```
int pid;
...
pid = fork();
if (0 != pid) {
    // code of the parent
    ...
} else {
    // code of the child
    ...
}
...
```

Using fork safely

```
int pid;
...
pid = fork();
if (0 != pid) {
    // code of the parent
    ...
} else {
    // code of the child
    ...
}
...
```

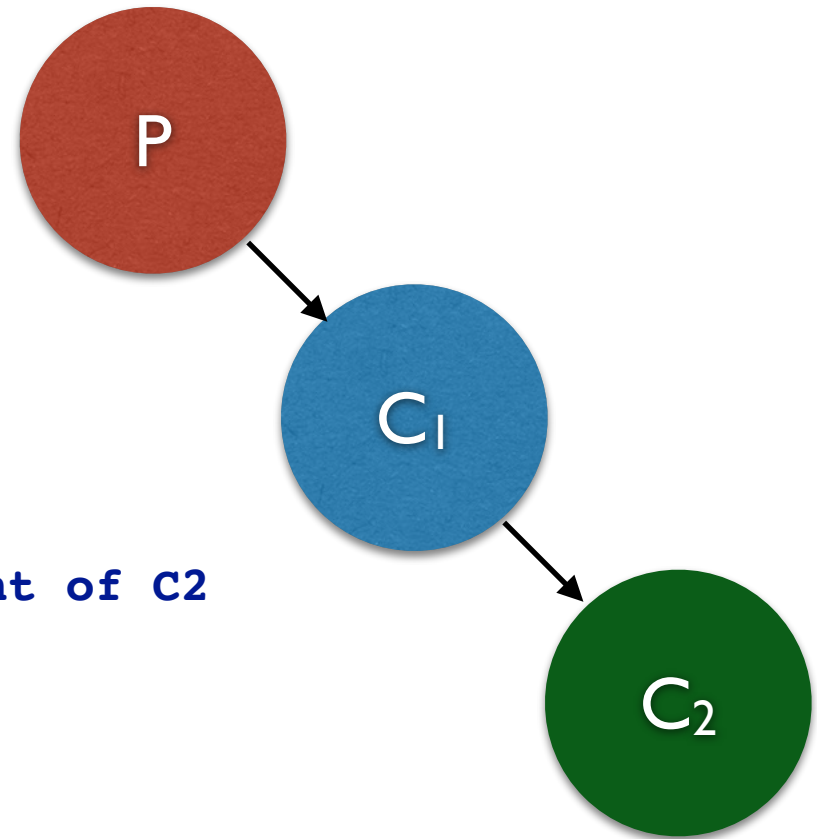
Using fork safely

```
int pid;
...
pid = fork();
if (0 != pid) {
    // code of parent P
    ...
} else {
    // code of child C
    ...
}
...
```



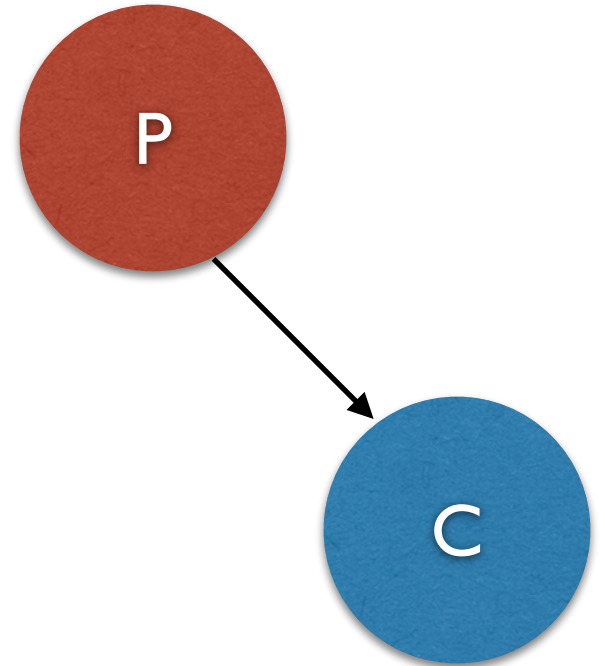
Using fork safely

```
int pid1, pid2;
pid1 = fork();
if (0 != pid1) {
    // code of parent P
    ...
} else {
    // code of child C1
    pid2 = fork();
    if (0 != pid2) {
        // code of child C1, parent of C2
        ...
    } else {
        // code of child C2
        ...
    }
}
```



Using fork even more safely

```
int pid;
pid = fork();
if (-1 == pid) {
    // error handling
    ...
} else if (0 != pid) {
    // code of parent P
    ...
} else {
    // code of child C
    ...
}
```



Joining *processes* in Unix

(Always RTFMP)

```
perrone@linuxremote1:~ (ssh)
WAIT(2) Linux Programmer's Manual WAIT(2)

NAME
    wait, waitpid, waitid - wait for process to change state

SYNOPSIS
    #include <sys/types.h>
    #include <sys/wait.h>

    pid_t wait(int *status);

    pid_t waitpid(pid_t pid, int *status, int options);

    int waitid(idtype_t idtype, id_t id, siginfo_t *infp, int options);

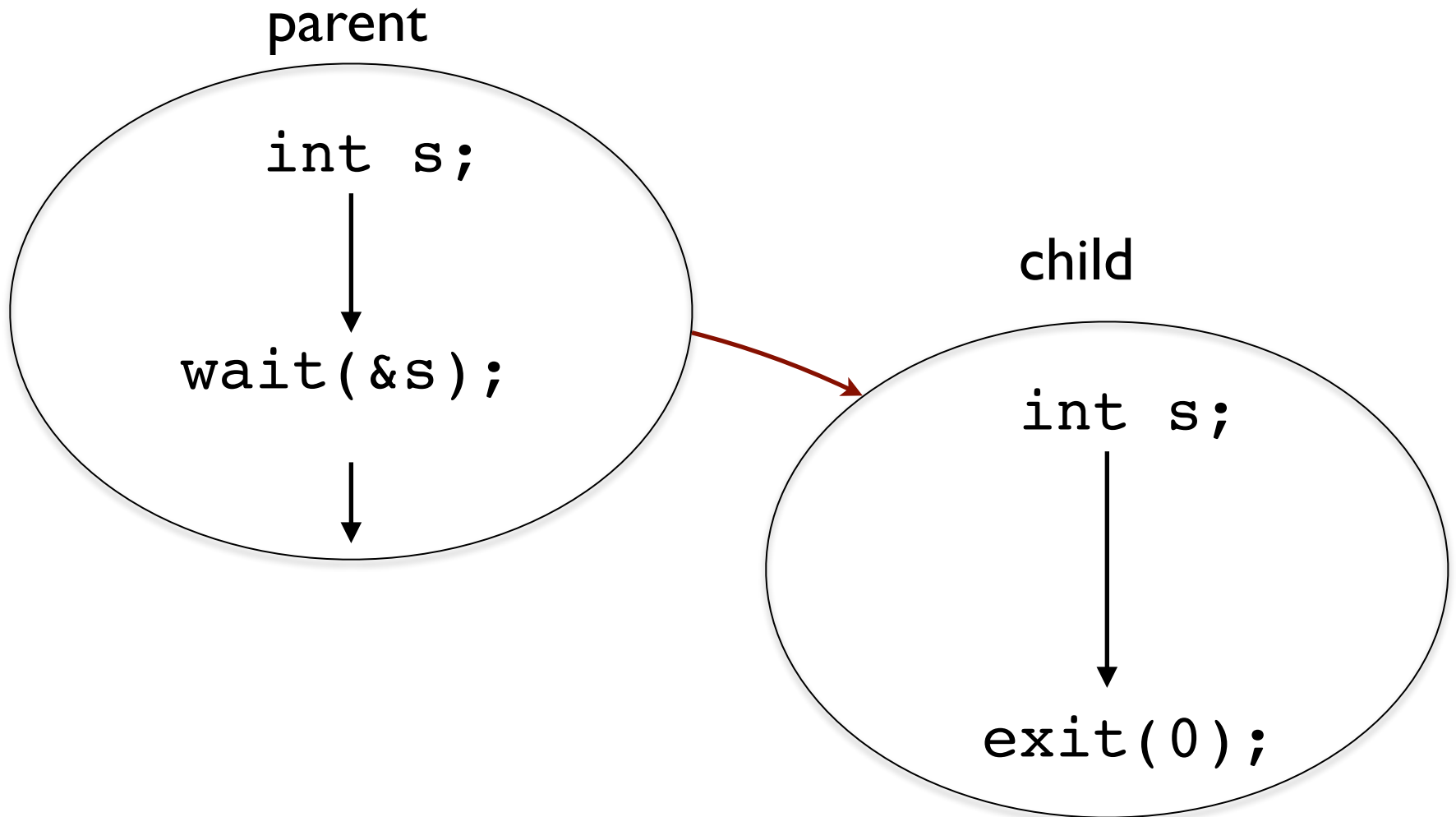
Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

waitid():
    _SVID_SOURCE || _XOPEN_SOURCE >= 500 || _XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED
    || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L

DESCRIPTION
    All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the sys-
    Manual page wait(2) line 1 (press h for help or q to quit)
```

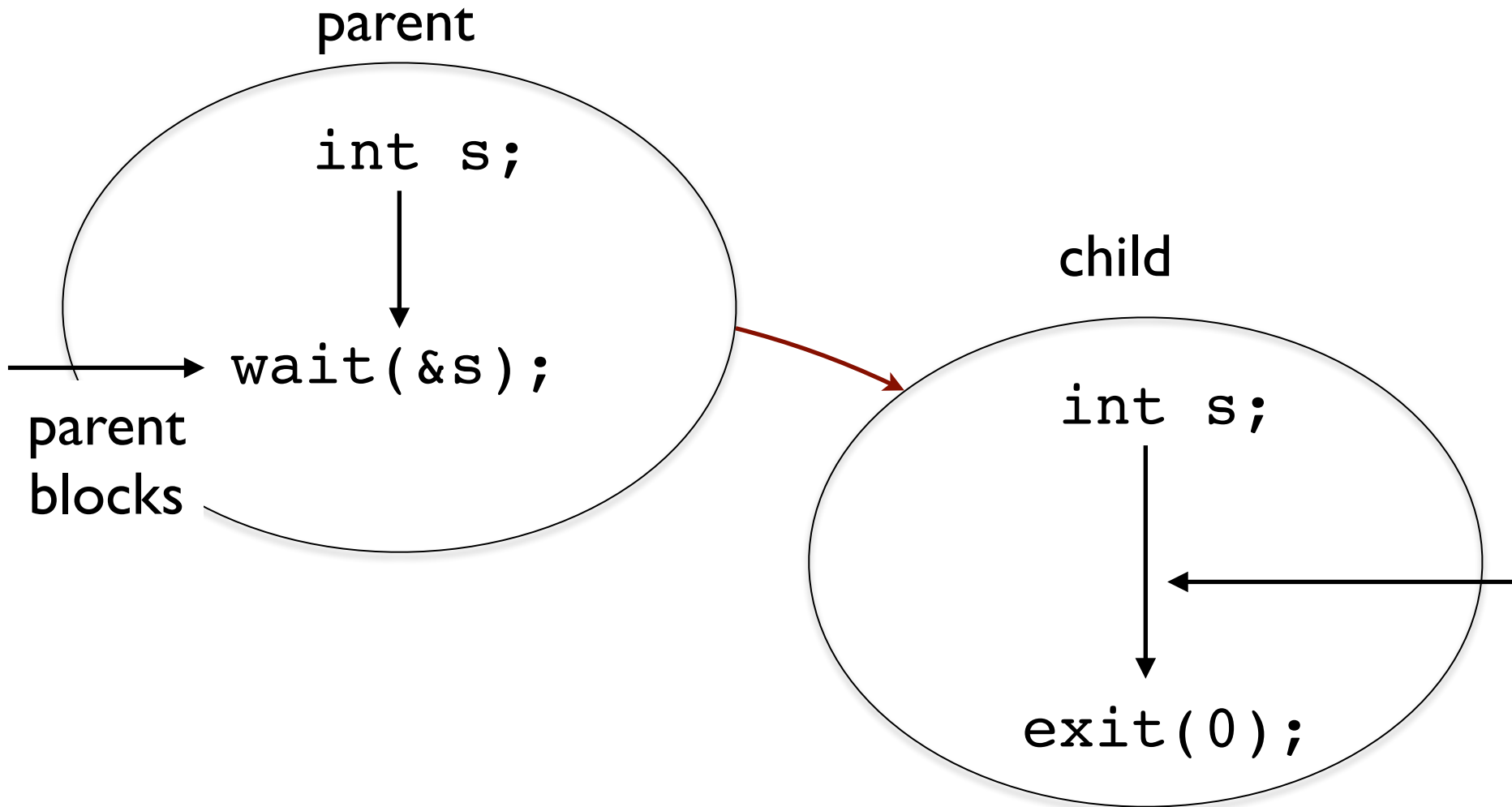
Waiting

(the inverse of forking)



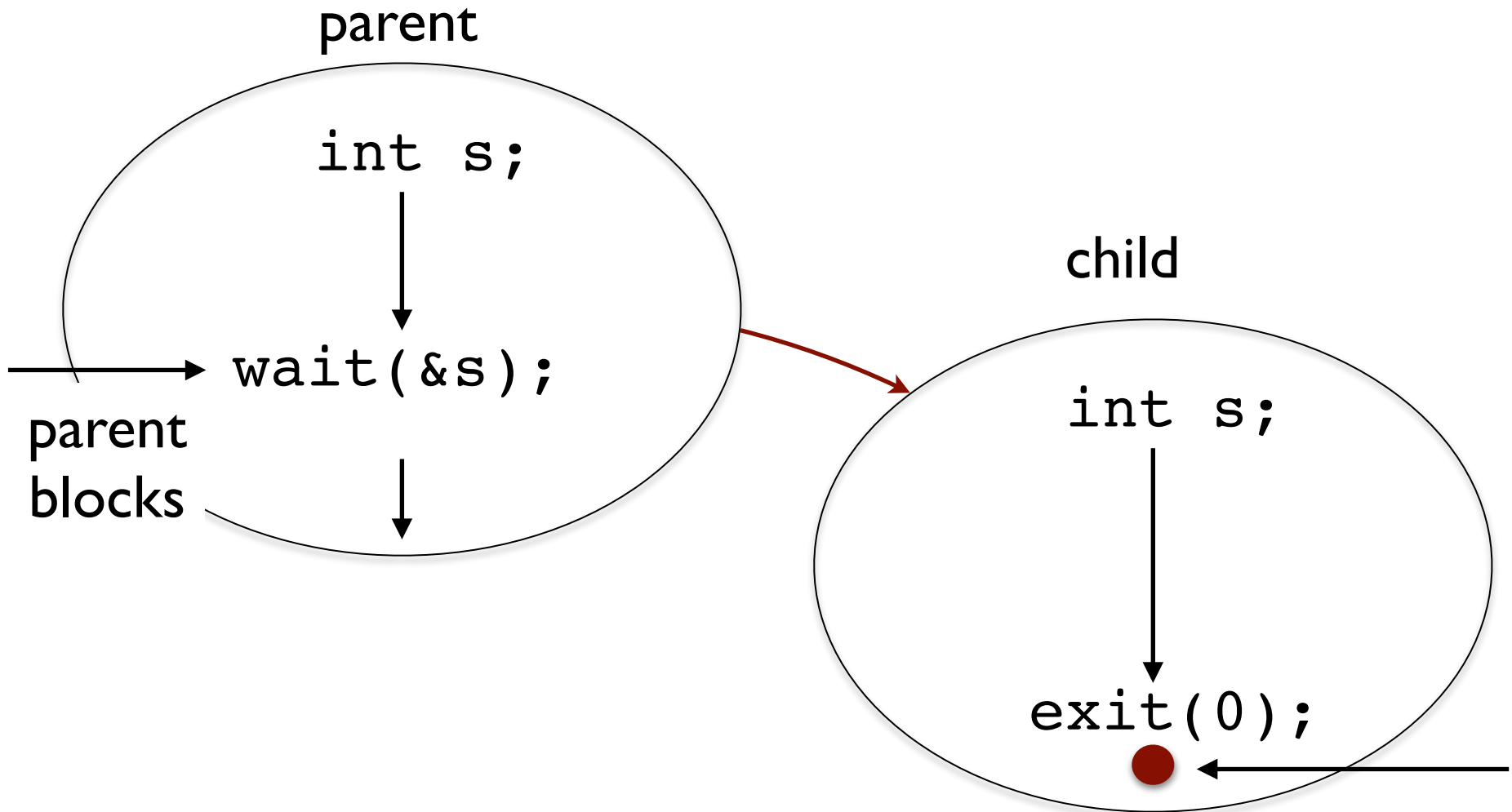
Waiting

(the inverse of forking)



Waiting

(the inverse of forking)



Waiting

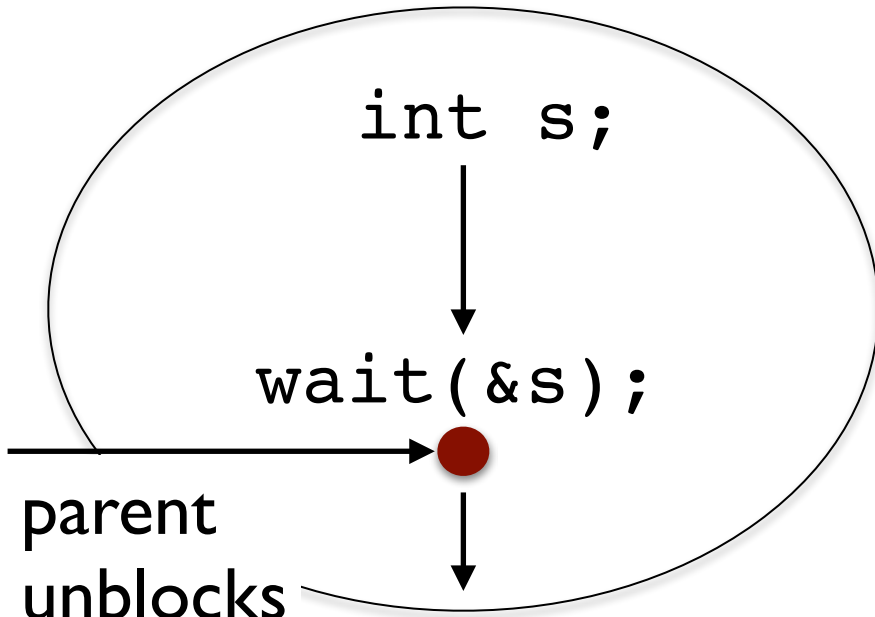
(the inverse of forking)

parent

```
int s;
```

```
wait(&s);
```

parent
unblocks



`fork(2)` and `exec(3)`

How do you run a process that has code (text) which is not identical to its parent's?

Linux Process Control

- `ps(1)`
- `top(1)` (MacOS X: make your terminal wide)
- `htop(1)`
- `pstree(1)`
- `kill(1)`

Just one more thing

Congratulations for getting to the end.

This video was a long journey.

But... before we go... one final piece of information.

Make note of it. It will be important for you to control the processes that you create by mistake.

kill(1)

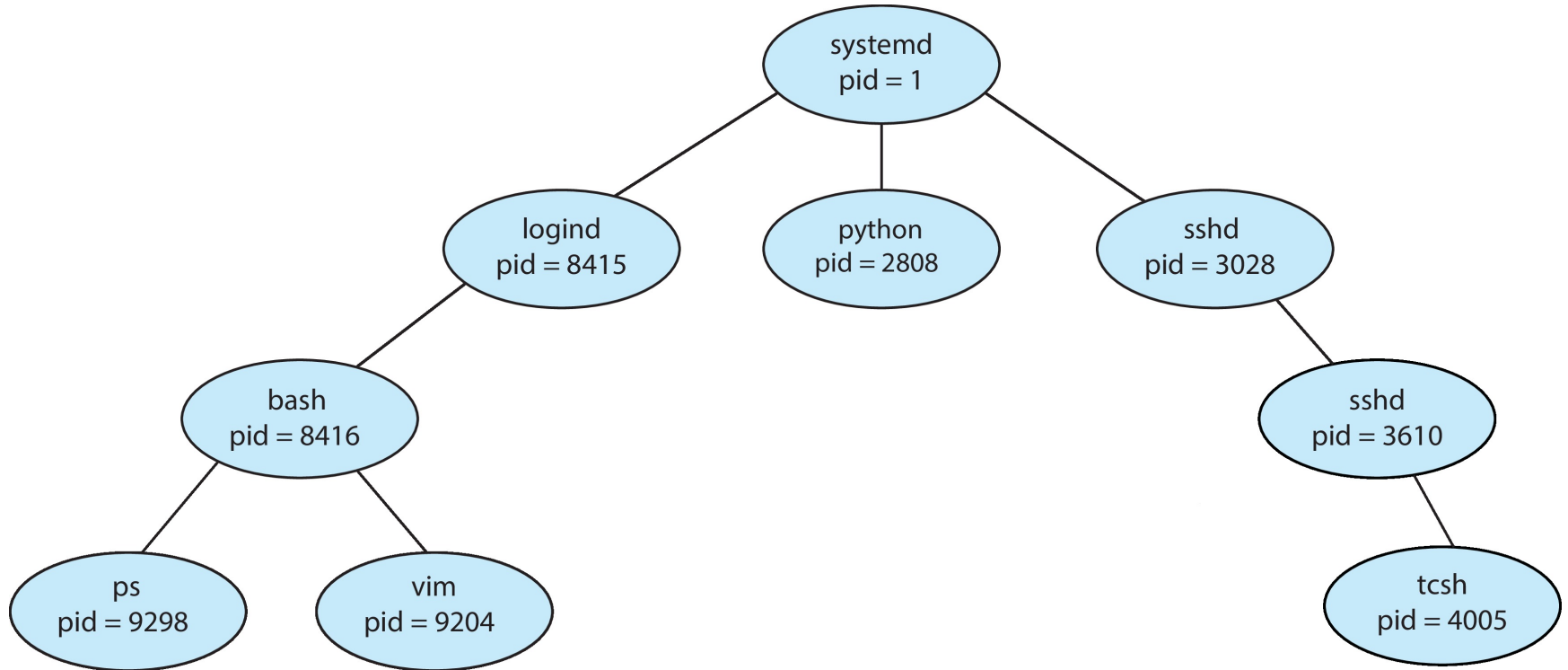
```
$ kill -9 12764
```

*Terminates the process with pid
12764*

```
$ kill -9 -1
```

*Terminates all processes run by you,
including your current shell.*

A Process Tree in Linux



Now, read its section I manual page and experiment with **ps**. Read the output carefully and pay attention at the columns to understand the wealth of information that comes back to you.

Process Control Block (PCB)

OS bookkeeping information associated with each process:

- Process state,
- Program counter,
- CPU registers,
- CPU scheduling information,
- Memory-management information,
- Accounting information,
- I/O status information,
-

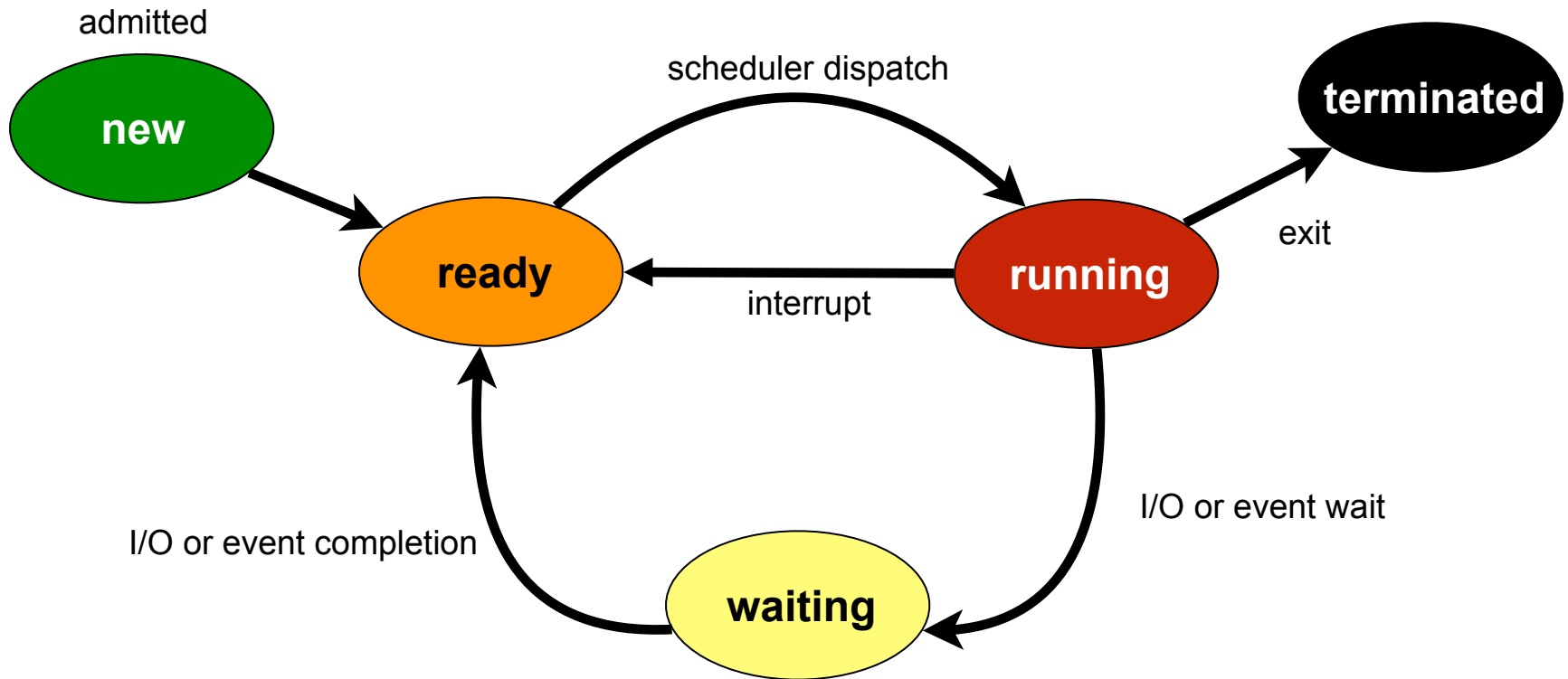
process id
process state
program counter
registers
memory limits
list of open files
⋮

Process State

As a process executes, it changes **state**:

- **new**: The process is being created.
- **running**: Instructions are being executed.
- **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 Transition Diagram

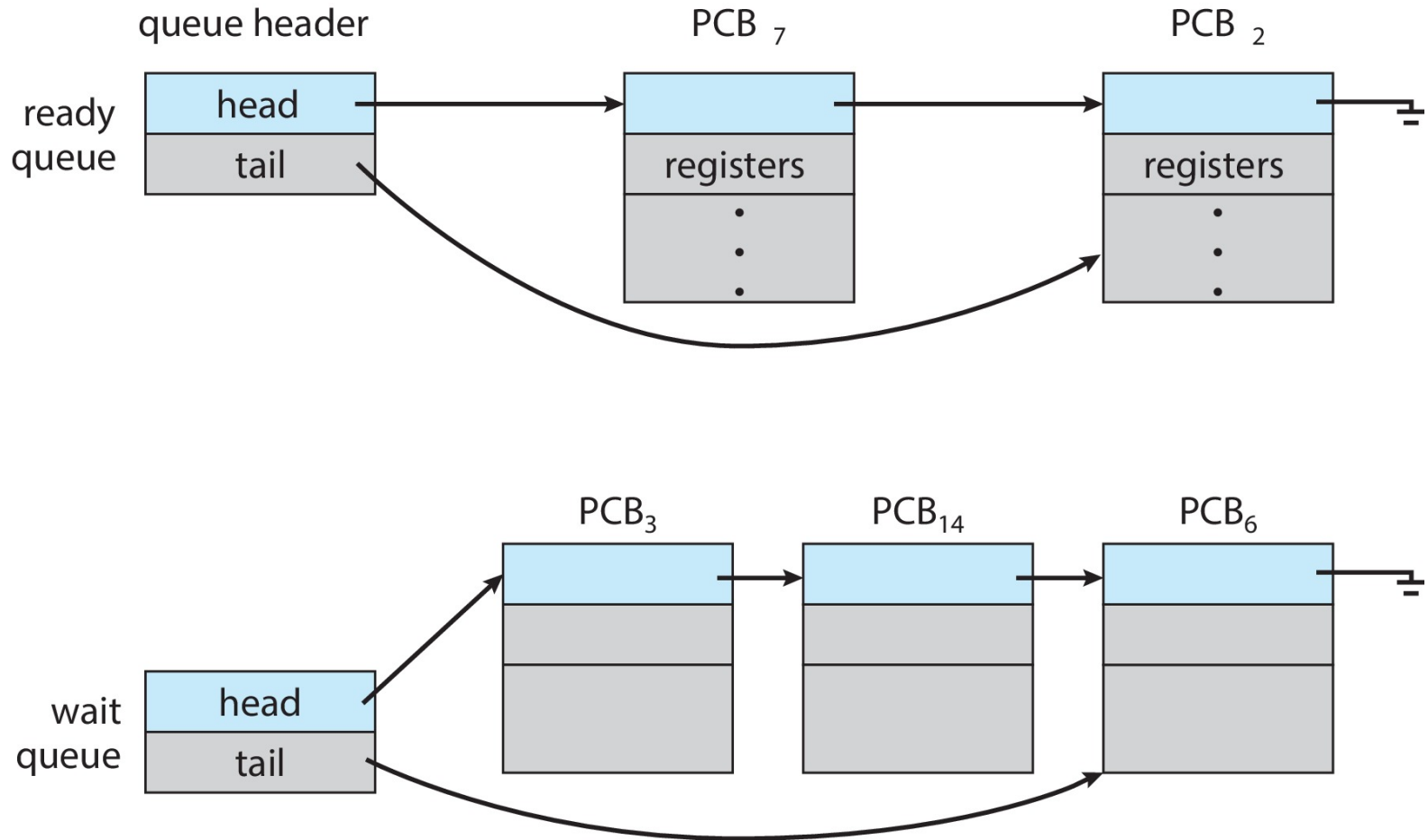


Process Scheduling Queues

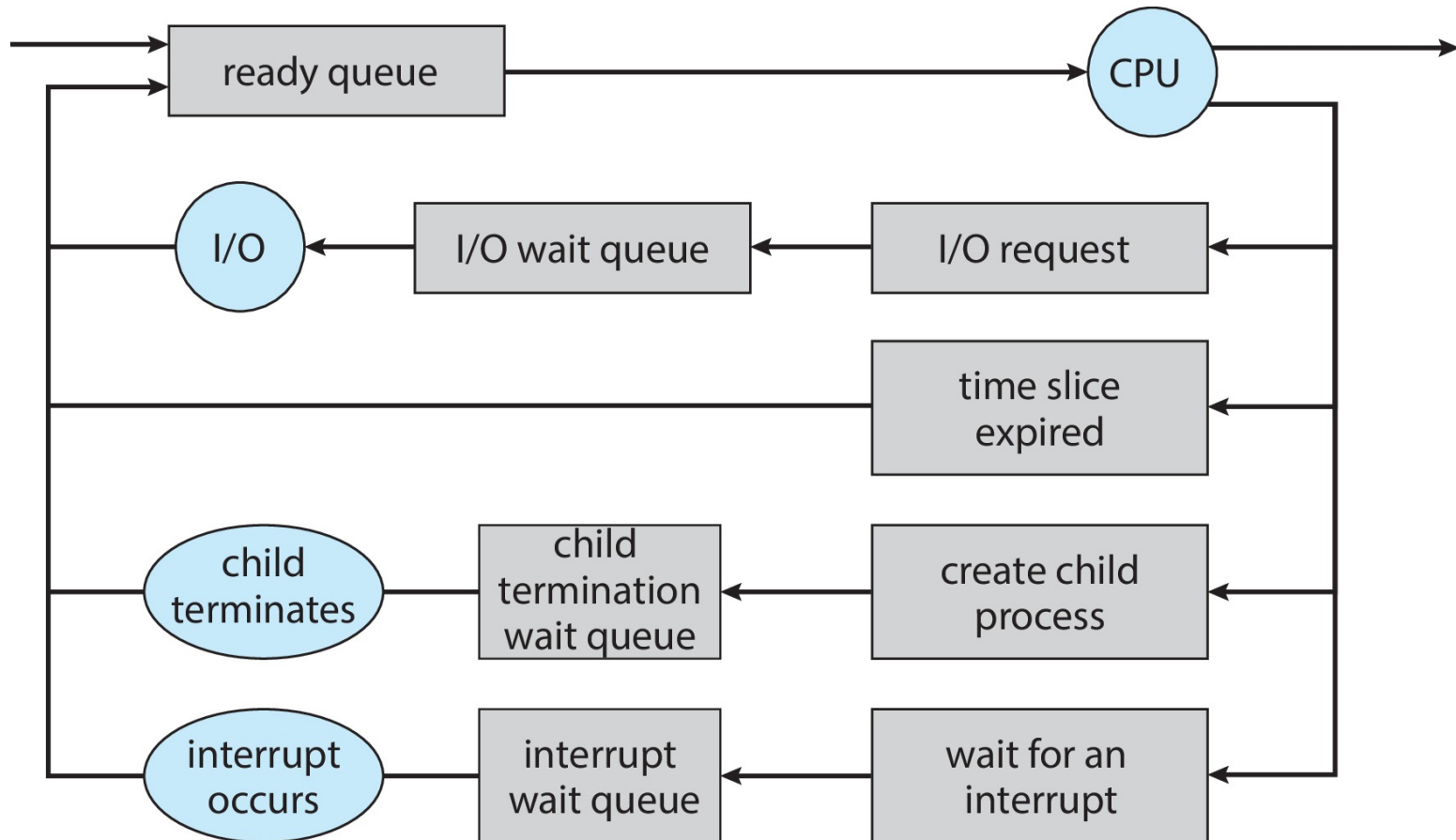
- **Job queue** – set of all processes in the system.
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute.
- **Device queues** – set of processes waiting for an I/O device.

Processes migrate between the various queues.

Processes and OS Queues



Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Schedulers

- **Long-term scheduler** is invoked very infrequently (seconds, minutes) \Rightarrow (may be slow; controls the *degree of multiprogramming*)
- **Short-term scheduler** is invoked very frequently (milliseconds) \Rightarrow (must be fast)

Processes can be described as either:

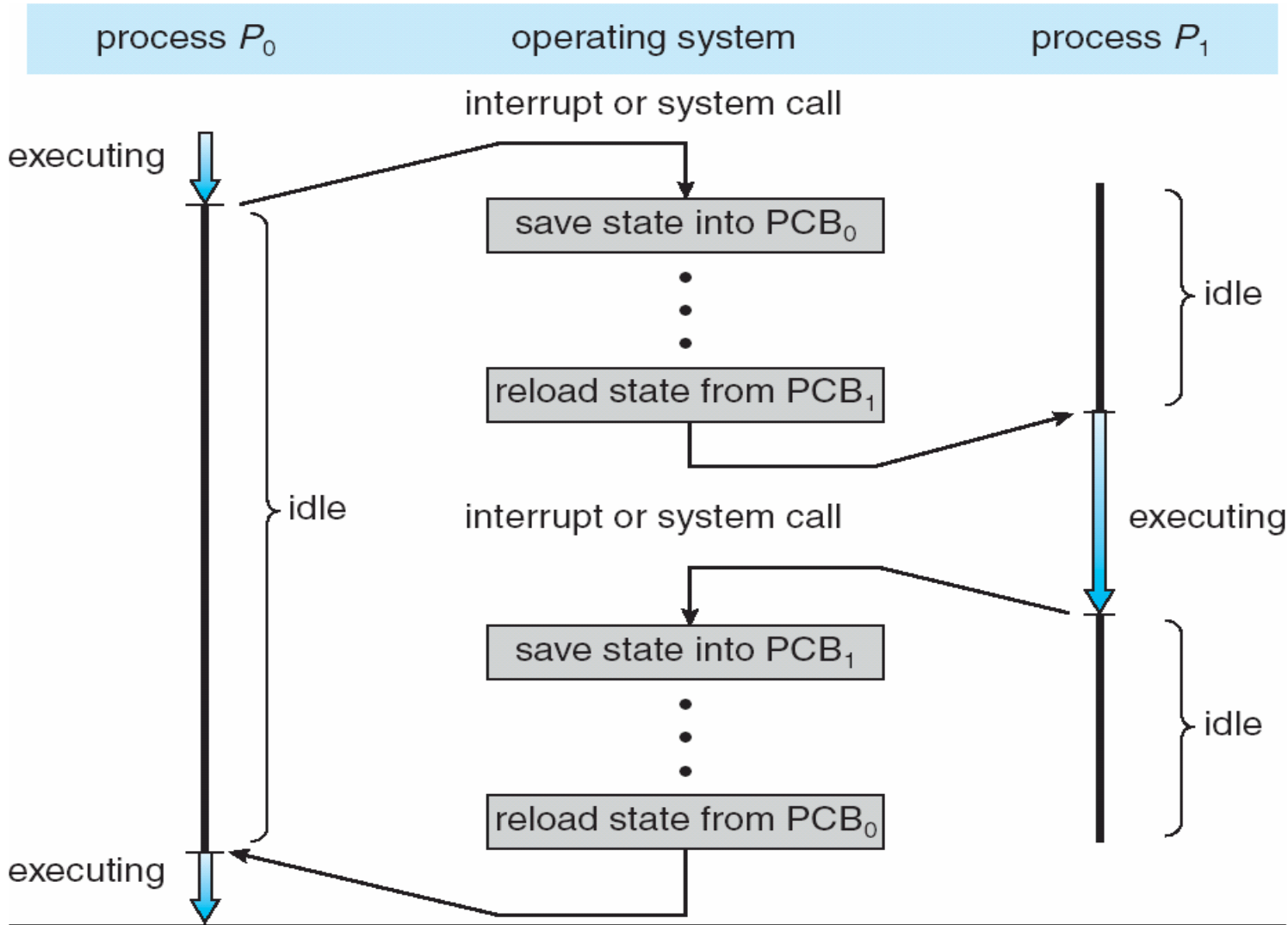
– **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts

– **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Context Switch

- 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.
- Context-switch time is **overhead**; the system does no useful work while switching.
- Time dependent on hardware support.

CPU Switching



Process Creation

Parent process creates children processes, which, in turn can create other processes, forming a *tree* of processes.

Resource sharing:

- Parent and children share all resources,
- Children share subset of parent's resources,
- Parent and child share no resources.

Execution:

- Parent and children execute concurrently,
- Parent may wait until children terminate.

Process Creation (Cont.)

- Address space:
 - Child has duplicate of parent's address space, or
 - Child can have a program loaded onto it.
- UNIX examples:
 - **fork** system call creates new process and returns with a pid (0 in child, > 0 in the parent),
 - **exec** system call can be used after a **fork** to replace the process' memory space with a new program.

Process Termination

Process executes last statement and asks the operating system to terminate it (**exit**)

- Output data from child to parent (via **wait**)
- Process' resources are deallocated by operating system

Parent may terminate execution of children processes (**abort**) if:

- Child has exceeded allocated resources,
- Task assigned to child is no longer required,
- If parent is exiting (some operating system do not allow child to continue if its parent terminates)
 - All children terminated - *cascading termination*

Cooperating Processes

- An ***independent*** process **cannot** affect or be affected by the execution of another process.
- A ***cooperating*** process **can** affect or be affected by the execution of another process.
- Advantages of process cooperation:
 - Information sharing,
 - Computation speed-up,
 - Modularity,
 - Convenience.

Interprocess Communication (IPC)

- 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*)
 - where *message* has fixed or variable size
- If processes *P* and *Q* wish to communicate, they need to:
 - establish a *communication link* between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., shared memory, hardware bus)
 - logical (e.g., logical properties)

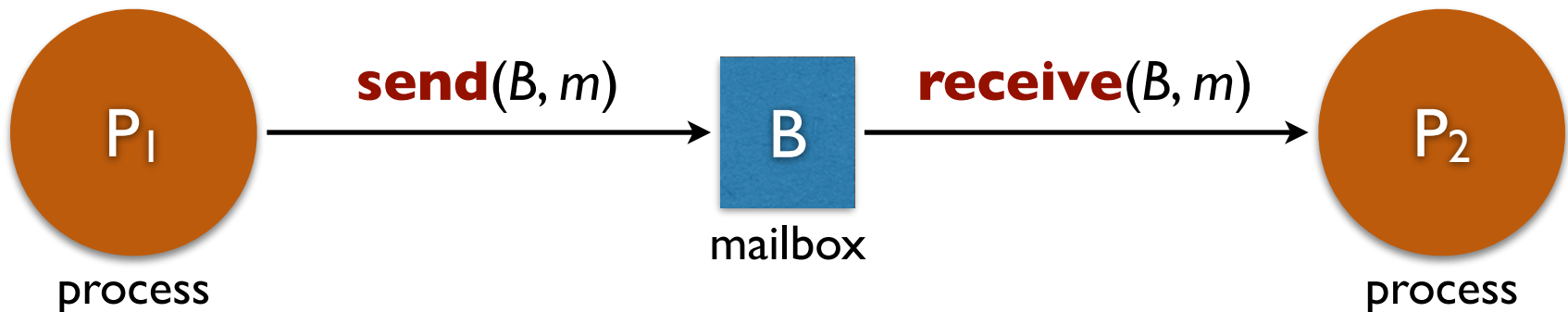
IPC Design Dimensions

(or *properties*)

- **Naming**
 - **direct or indirect**
 - **symmetric or asymmetric**
- **Synchronization**
 - **blocking send, non-blocking send**
 - **blocking receive, non-blocking receive**
- **Buffering**
 - **zero capacity**
 - **bounded capacity**
 - **unbounded capacity**

Indirect Communication

- 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



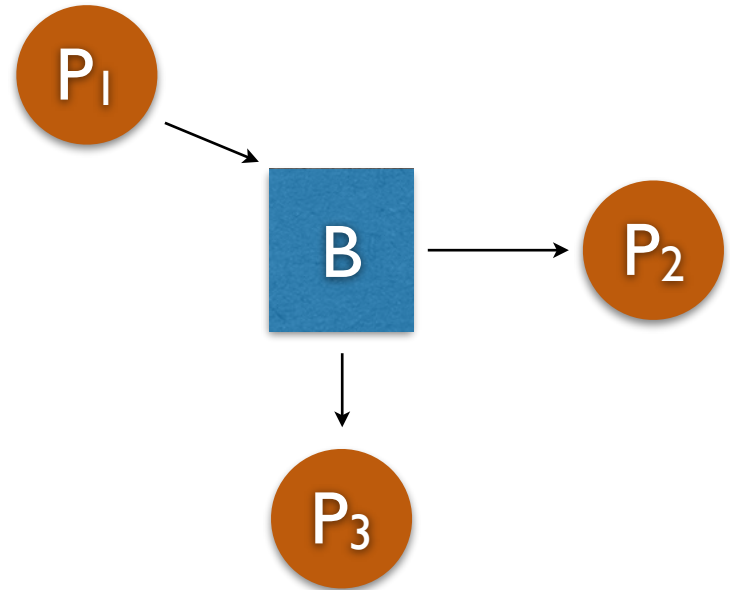
Indirect Communication

- Operations:
 - create a new mailbox,
 - 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 .

Indirect Communication

- Mailbox sharing

- P_1 sends **one message m**
- P_2 and P_3 try to receive
- Who gets the message?



- Solutions

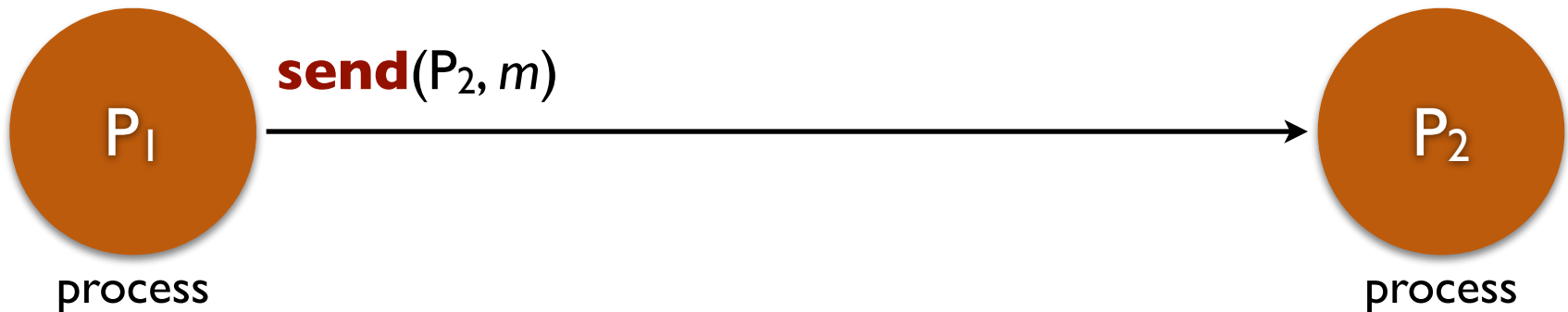
- Allow a link to be associated with at most two processes.
- Allow only one process at a time to execute a receive operation.
- Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Direct Communication

Processes must name each other explicitly:

send ($P, message$) – send a message to process P

receive($Q, message$) – receive a message from process Q

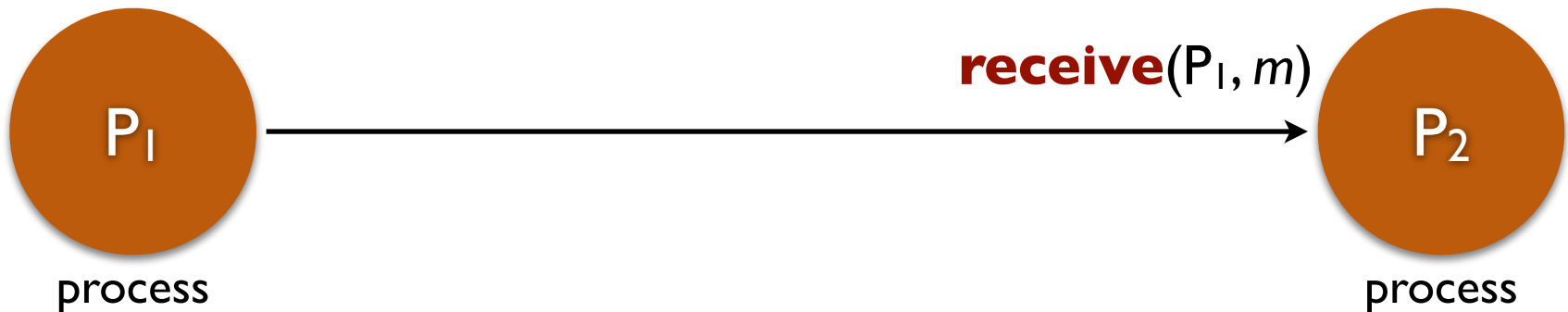


Direct Communication

Processes must name each other explicitly:

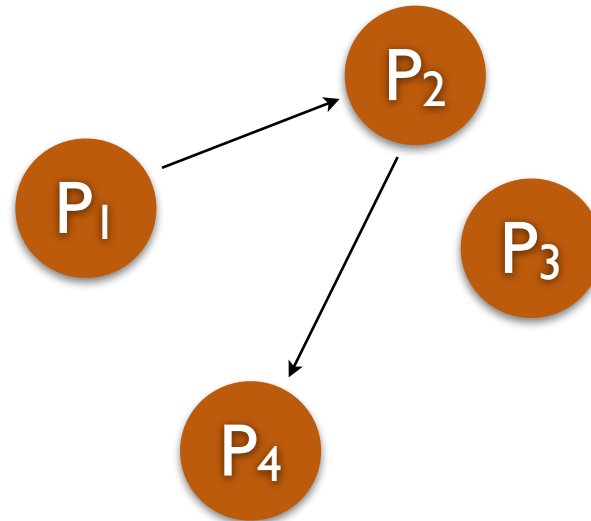
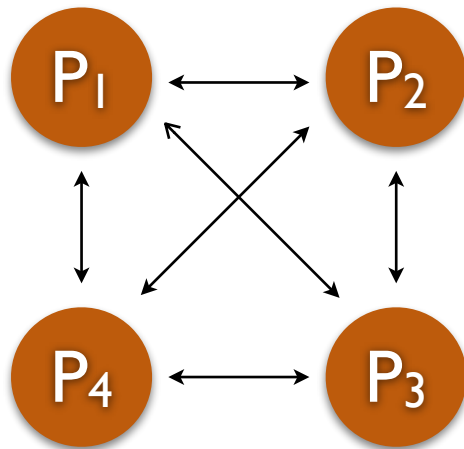
send ($P, message$) – send a message to process P

receive($Q, message$) – receive a message from process Q



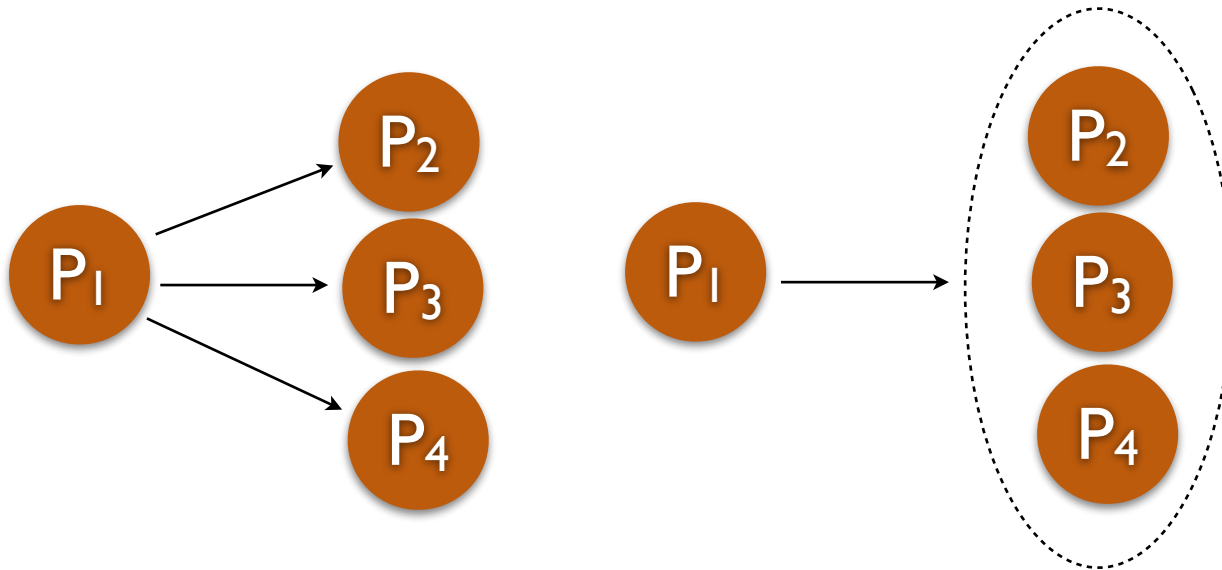
Direct Communication

- Properties of communication links
 - **Links may be established automatically or may have to be created explicitly**
 - Links may be point-to-point or point-to-multipoint
 - Links may be unidirectional or bidirectional



Direct Communication

- Properties of communication links
 - Links may be established automatically or may have to be created explicitly
 - **Links may be point-to-point or point-to-multipoint**
 - Links may be unidirectional or bidirectional



Direct Communication

- Properties of communication links
 - Links may be established automatically or may have to be created explicitly
 - Links may be point-to-point or point-to-multipoint
 - **Links may be unidirectional or bidirectional**



Synchronization

- *Message passing* may be either blocking or non-blocking.
- **Blocking** is considered **synchronous**:
 - **Blocking send** has the sender block until the message is received.
 - **Blocking receive** has the receiver block until a message is available.
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** has the sender send the message and continue.
 - **Non-blocking receive** has the receiver receive a valid message or null.

Buffering

Queue of messages attached to the link;
implemented in one of three ways:

1. Zero capacity – 0 messages

Sender must wait for receiver (rendezvous).

2. Bounded capacity – finite length of n messages.

Sender must wait if link full.

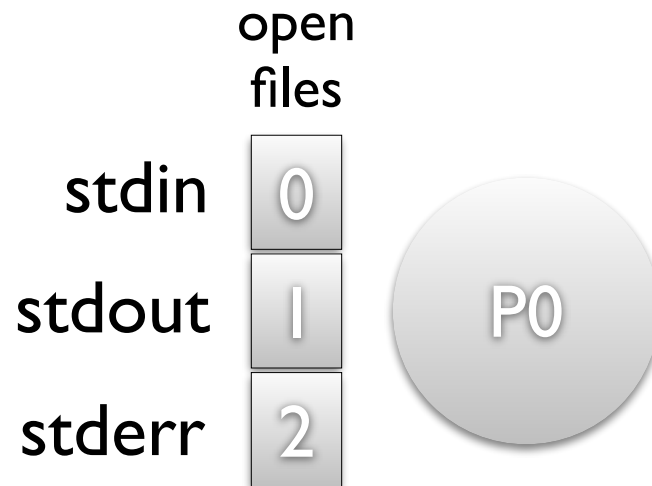
3. Unbounded capacity – infinite length. Sender never waits.

Unix pipe(2)

- Point to point
- Unidirectional
- For processes related by birth (same machine)
- Reliable delivery
- Stream of bytes
- FIFO
- Virtually identical to reading and writing to a file (low level file I/O)

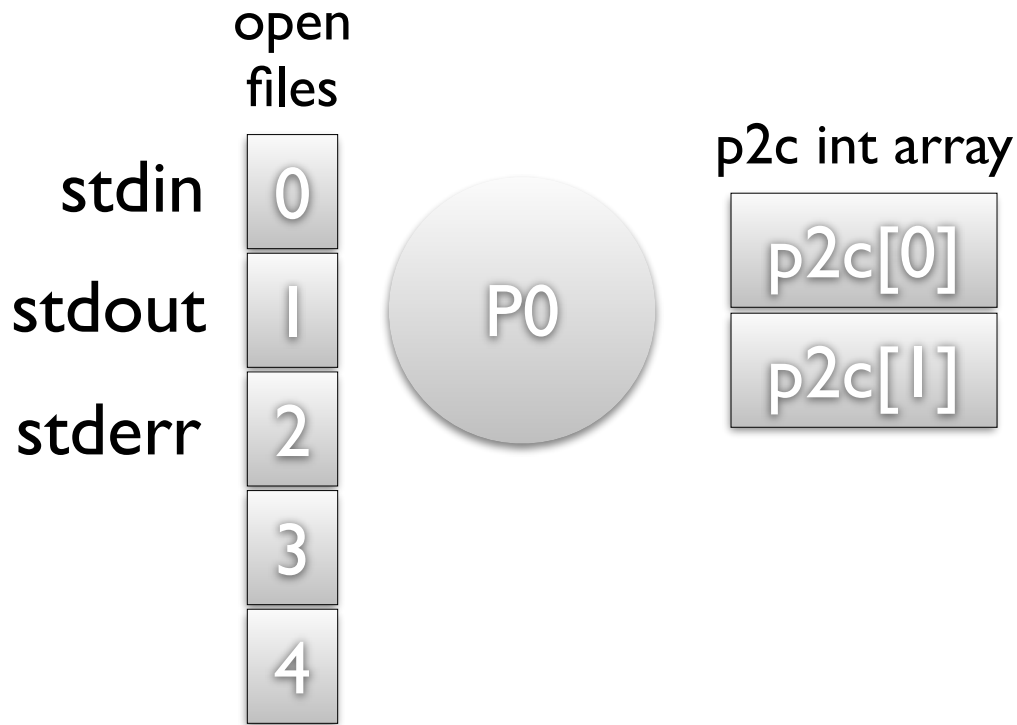
Unix pipe(2)

A process P0 is born

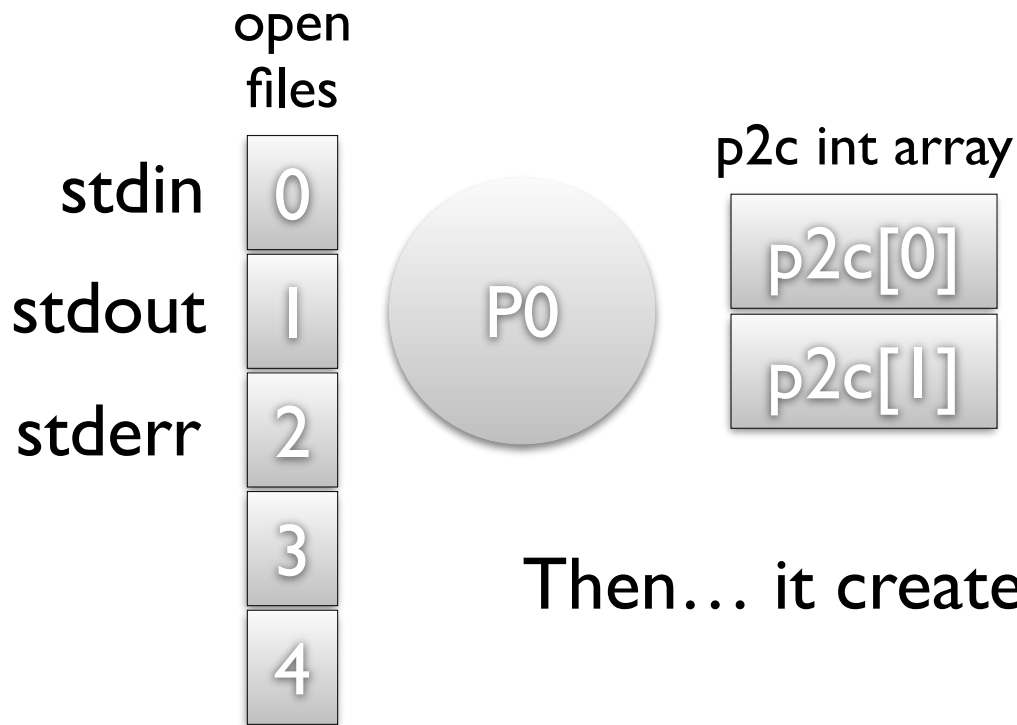


Before creating a child with whom it will communicate, it creates a pipe (system call).

Unix pipe(2)

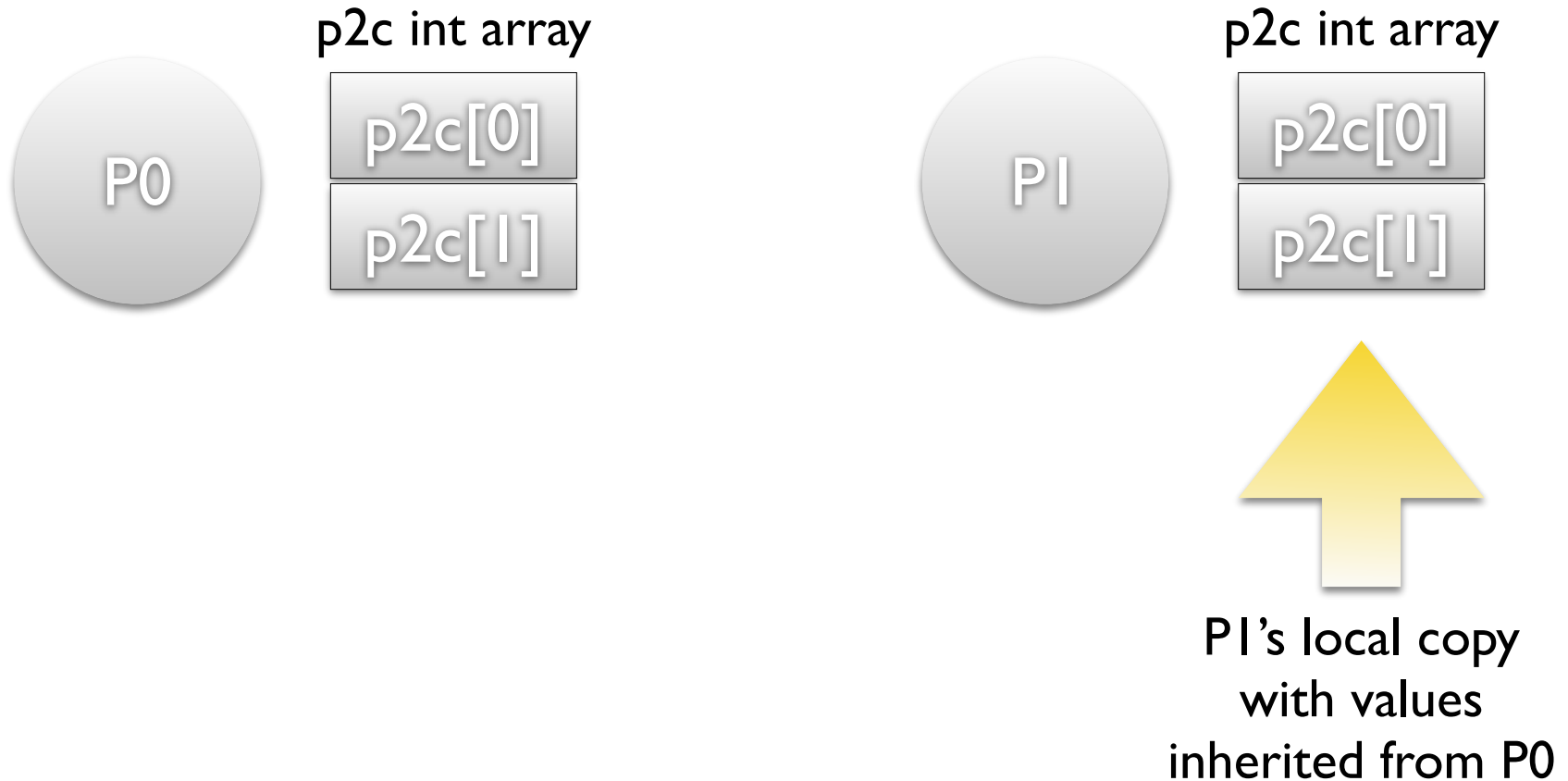


Unix pipe(2)

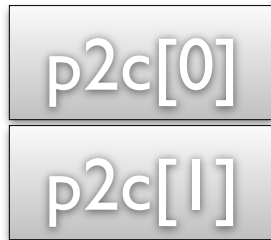


Then... it creates child P1 with **fork**

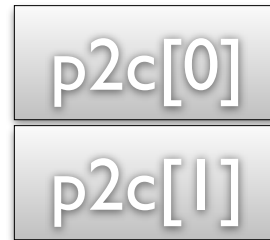
Unix pipe(2)



Unix pipe(2)

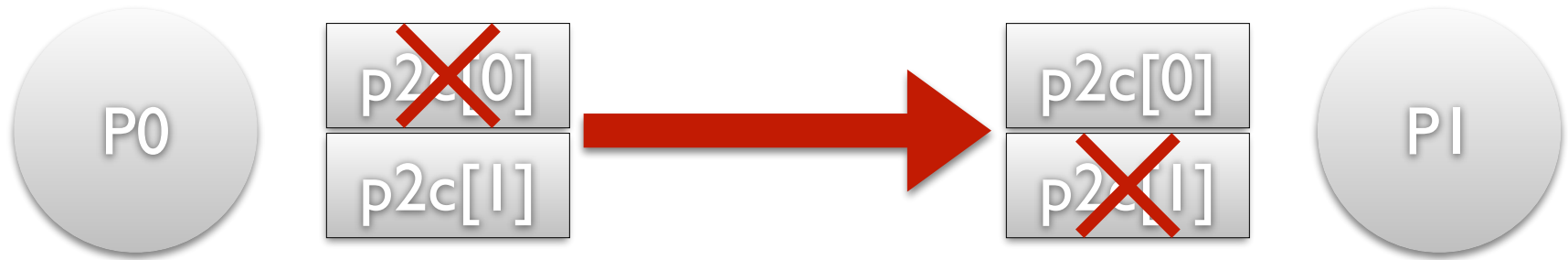


P0 closes the input end of the pipe (index 0)



P1 closes the output end of the pipe (index 1)

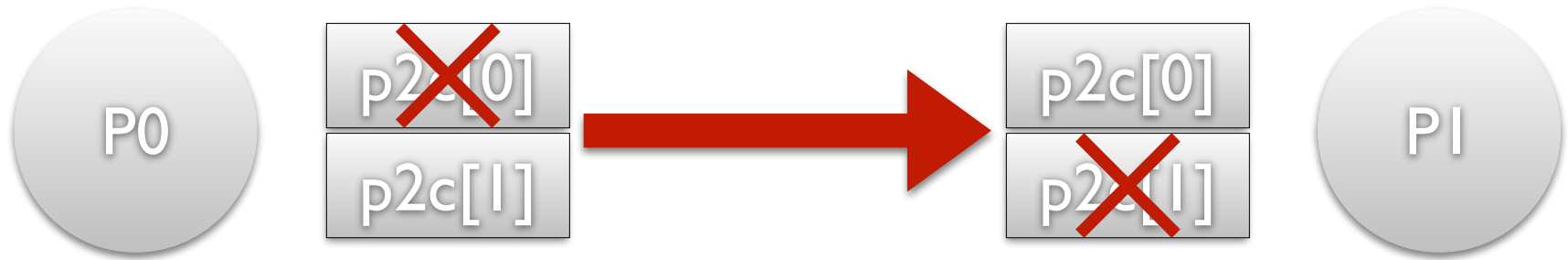
Unix pipe(2)



P0 closes the input end of the pipe (index 0)

P1 closes the output end of the pipe (index 1)

Unix pipe(2)



P0 writes to file
descriptor p2c[1]

`write(2)`

P1 reads from file
descriptor p2c[0]

`read(2)`

IPC Mechanisms

- File
- Pipe
- Named pipe
- Shared memory
- Message passing
- Mailbox
- Remote procedure calls
- Sockets (TCP, datagram)

- What are the properties of each?
- What are the advantages and disadvantages of each?
- How do you select one to use?

IPC Mechanisms

- File
- Pipe
- Named pipe
- **Shared memory**
- Message passing
- Mailbox
- Remote procedure calls
- Sockets (TCP, datagram)

- What are the properties of each?
- What are the advantages and disadvantages of each?
- How do you select one to use?