Operating Systems

2. Process Concept

Process: The Concept

- Process = A program in execution
  - Execution progress in a sequential fashion
- Example processes
  - OS kernel
  - OS shell
  - Program executing after compilation
  - www-browser
- Process management by OS
  - Allocate resources
    - ➢ Processor, main memory, I/O modules, timers, …
  - Schedule: Interleave their execution
    - ➢ Watch for processor utilization, response time
  - Inter-process communication, synchronization
    - ➢ Check potential deadlocks
    - ➢ Interleaving and non determinism imply increased difficulties!

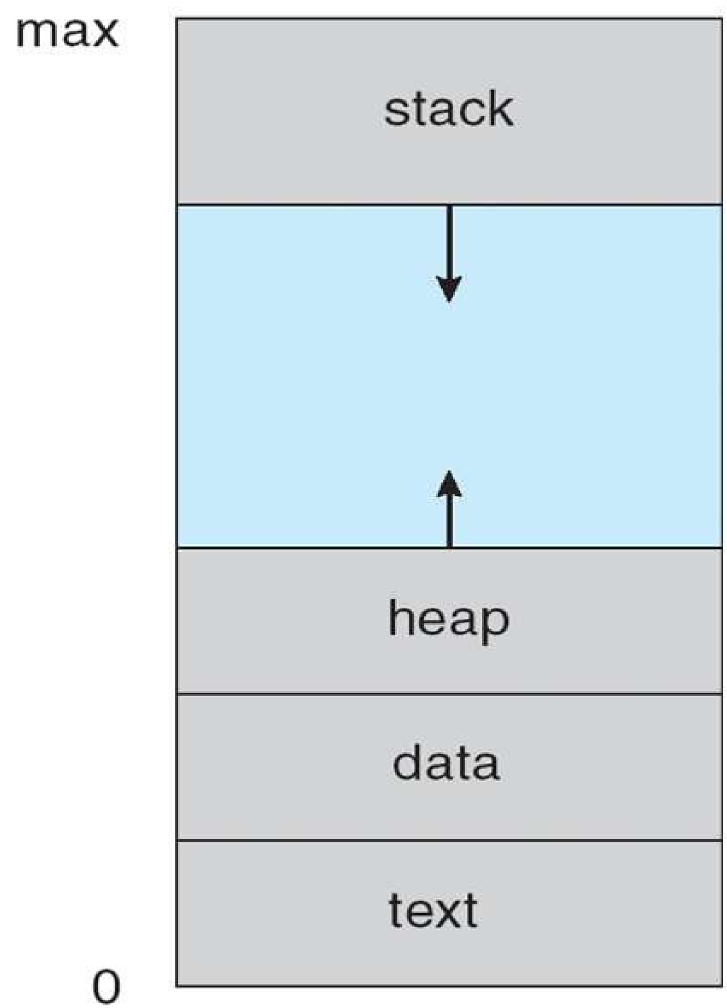
The Process

A process consists of multiple parts

- Program code, also called text section
- Current activity including program counter, processor registers
- Stack containing temporary data
  - Function parameters, return addresses, local variables
- Data section containing global variables
- Heap containing memory dynamically allocated during run time

Process Address Space

- A list of memory locations from some min (usually 0) to some max that a process can read and write



Relation Between Program And Process

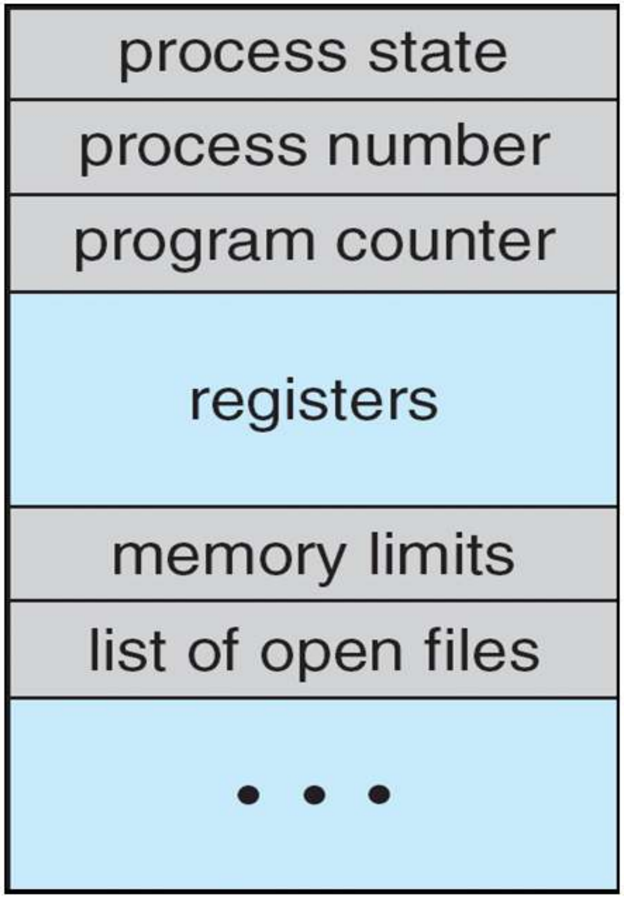
- Program is passive entity, process is active
  - Program: Static code + static data
  - Process: Dynamic instantiation of code + data + more

- Program becomes process when executable file loaded into memory

- No 1 to 1 mapping between program and process
  - One program may run many processes
    - Separate execution sequences
  - Multiple users may execute the same program
    - Text section is equivalent

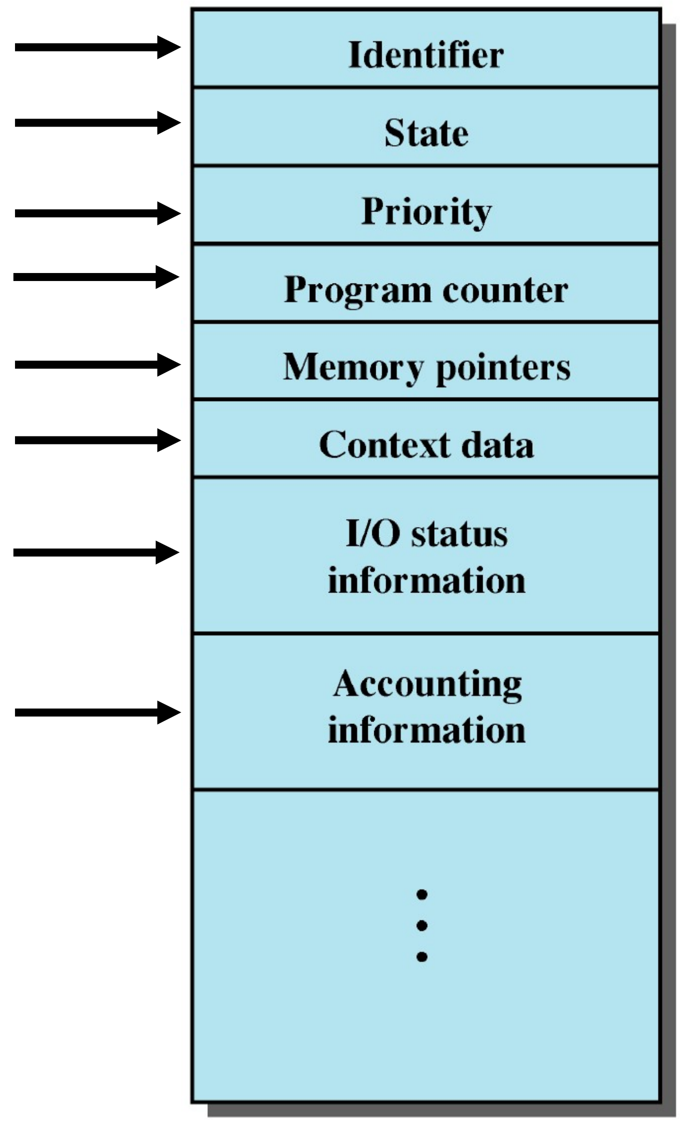
Process Control Block

- OS keeps all the data it needs about a process in the process control block (PCB)
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information



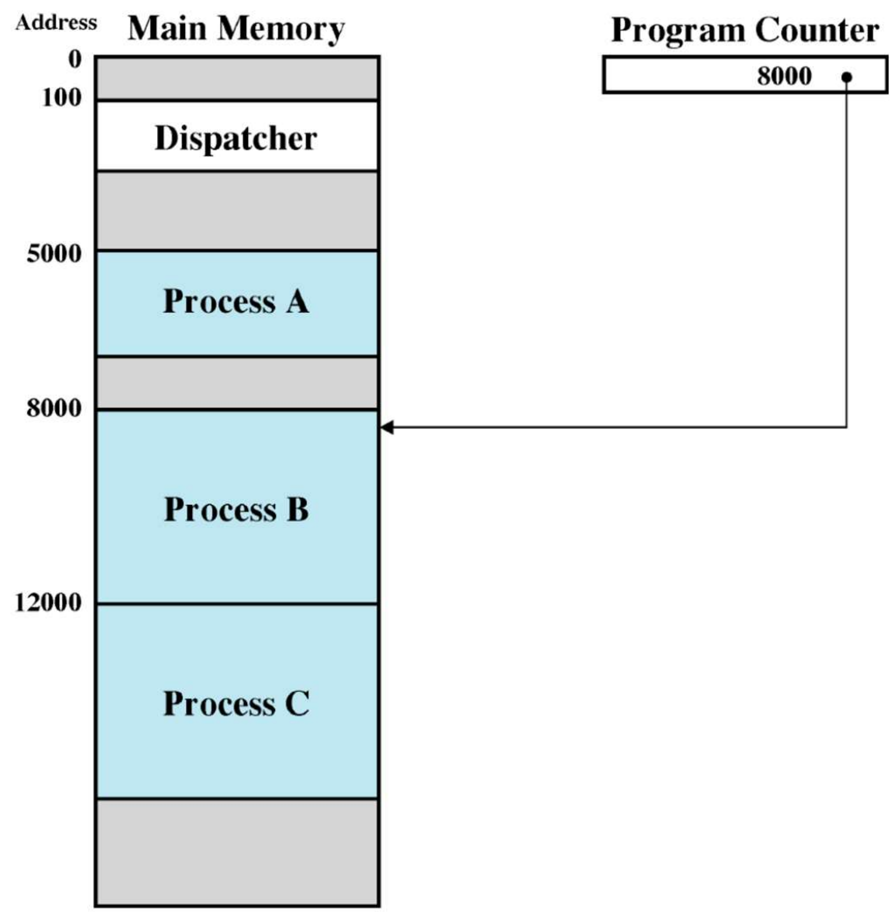
Process Control Block

- Distinction between processes
- Knowledge on processes ready to execute
- Discriminate between processes
- Know the next instruction to execute
- Find the program and data stored in memory
- Keep information about registers
- Keep track about resources, e.g., I/O devices
- Collect information on utilization of the system



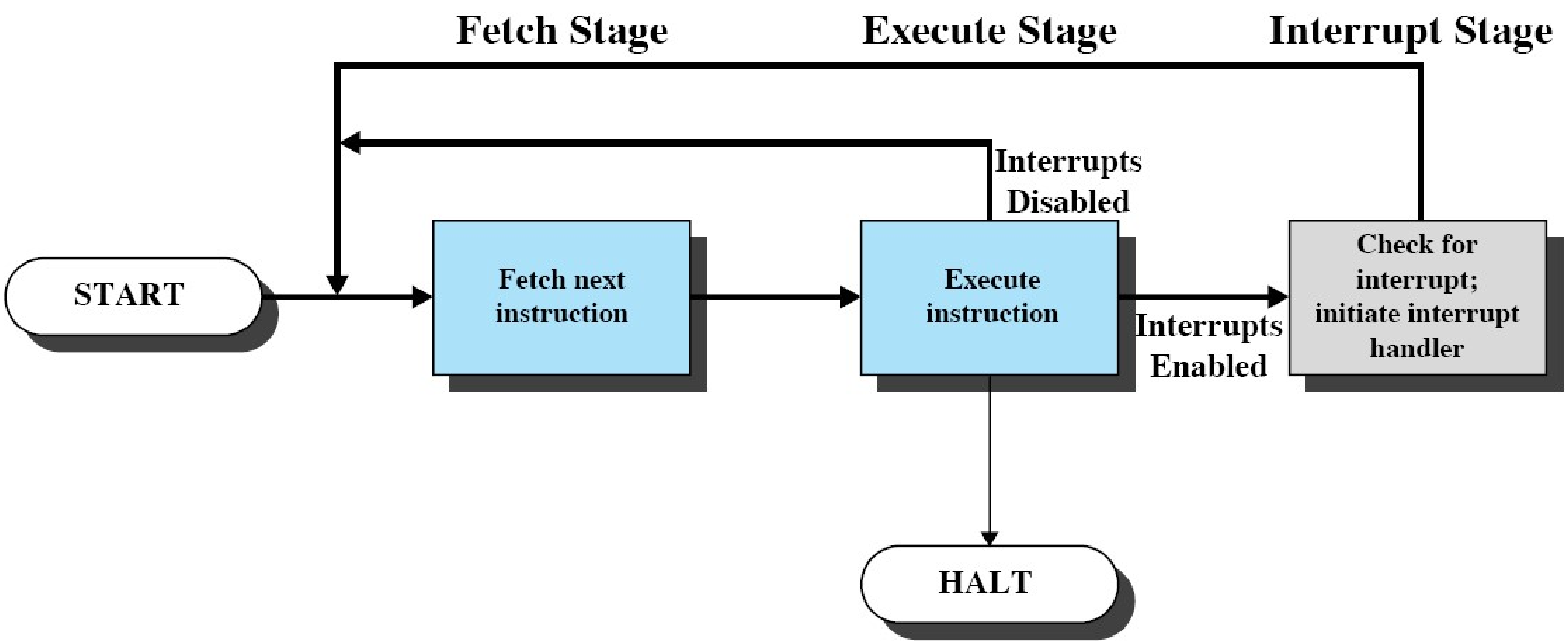
Dispatcher/Scheduler

- OS program that decides which process to run next
- Uses Process Block Control (PBC) information
  - Decision what is the next process to execute



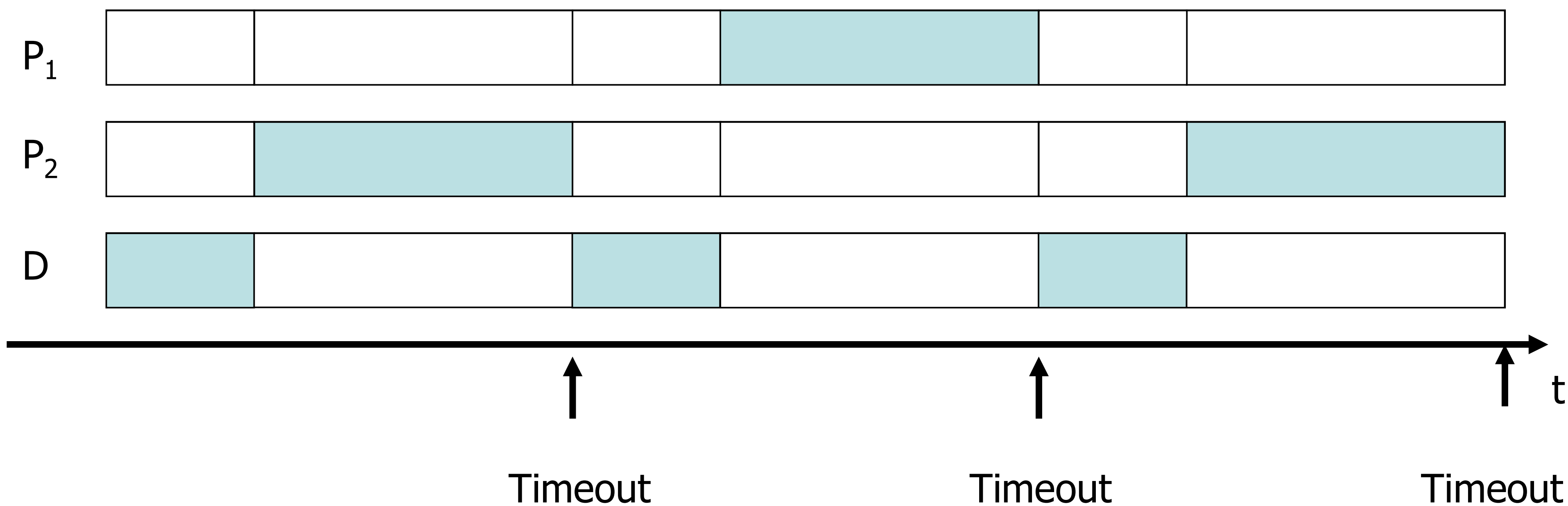
How Does the Dispatcher Gains Control?

- Register interrupt handlers
  - Timeout
  - I/O request



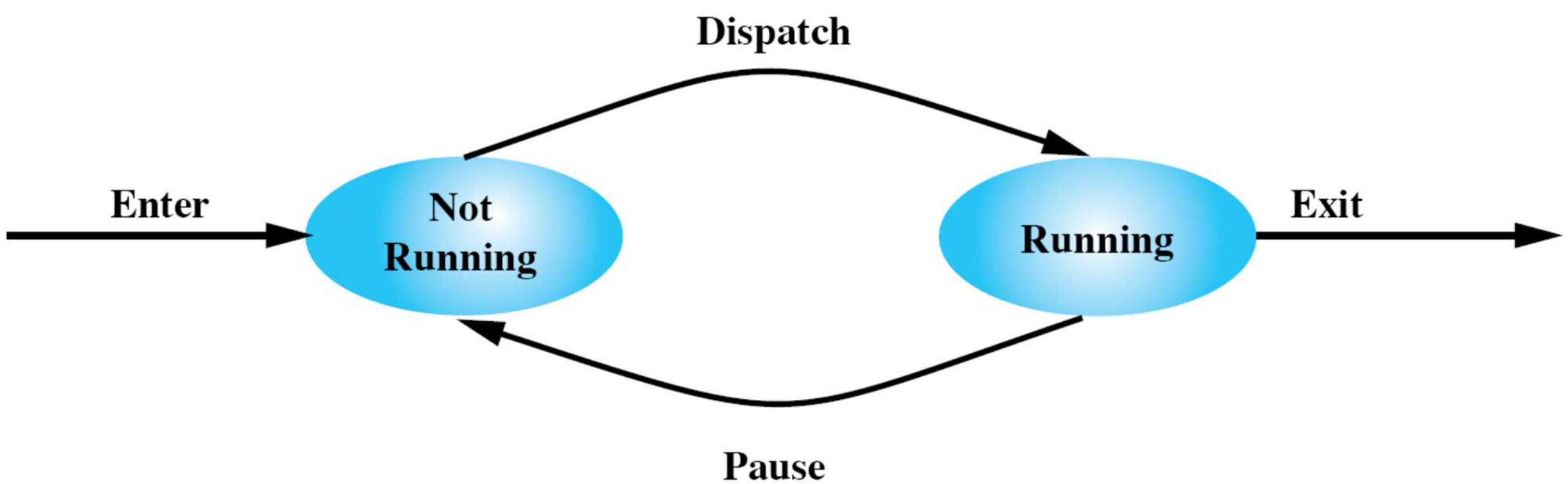
Execution with Timeouts

- On timeout dispatcher gains control
- Decides which process to execute next
- Dispatcher should perform as few instructions as possible



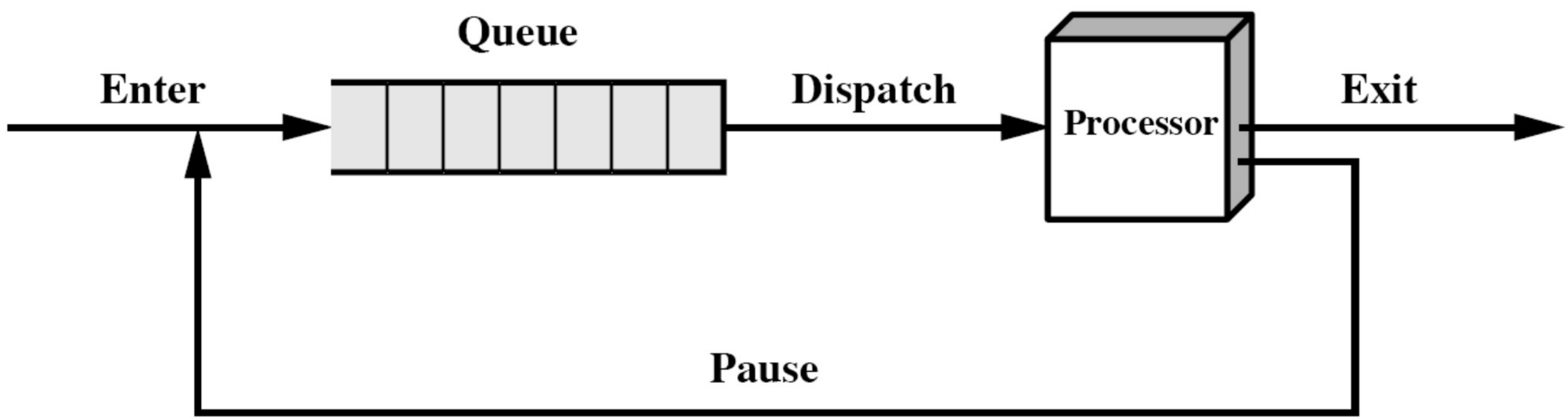
Two-State Process Model

- At any time a process is either being executed by a processor or not



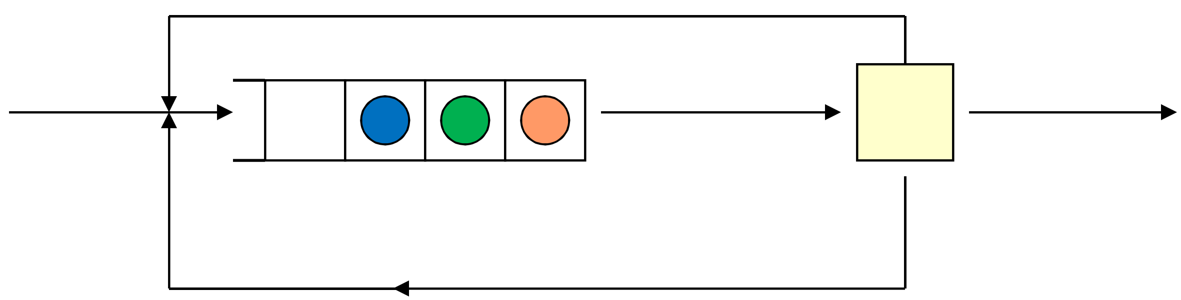
How to Use States for Scheduling?

- Queue of not-running processes
- Queues
  - FIFO, Priority Queue
- Design goal
  - Execute only few operations

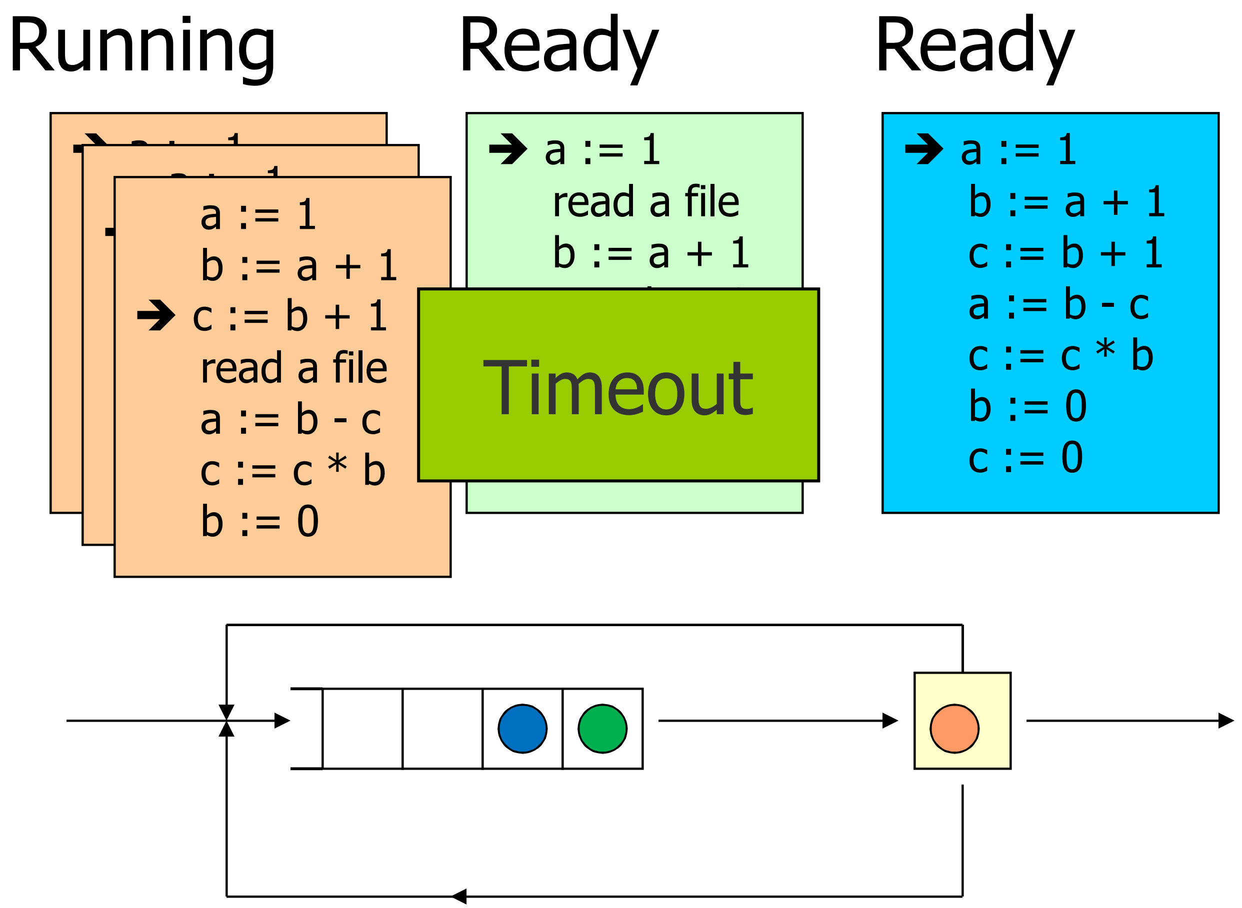


Is Two Process Model Sufficient?

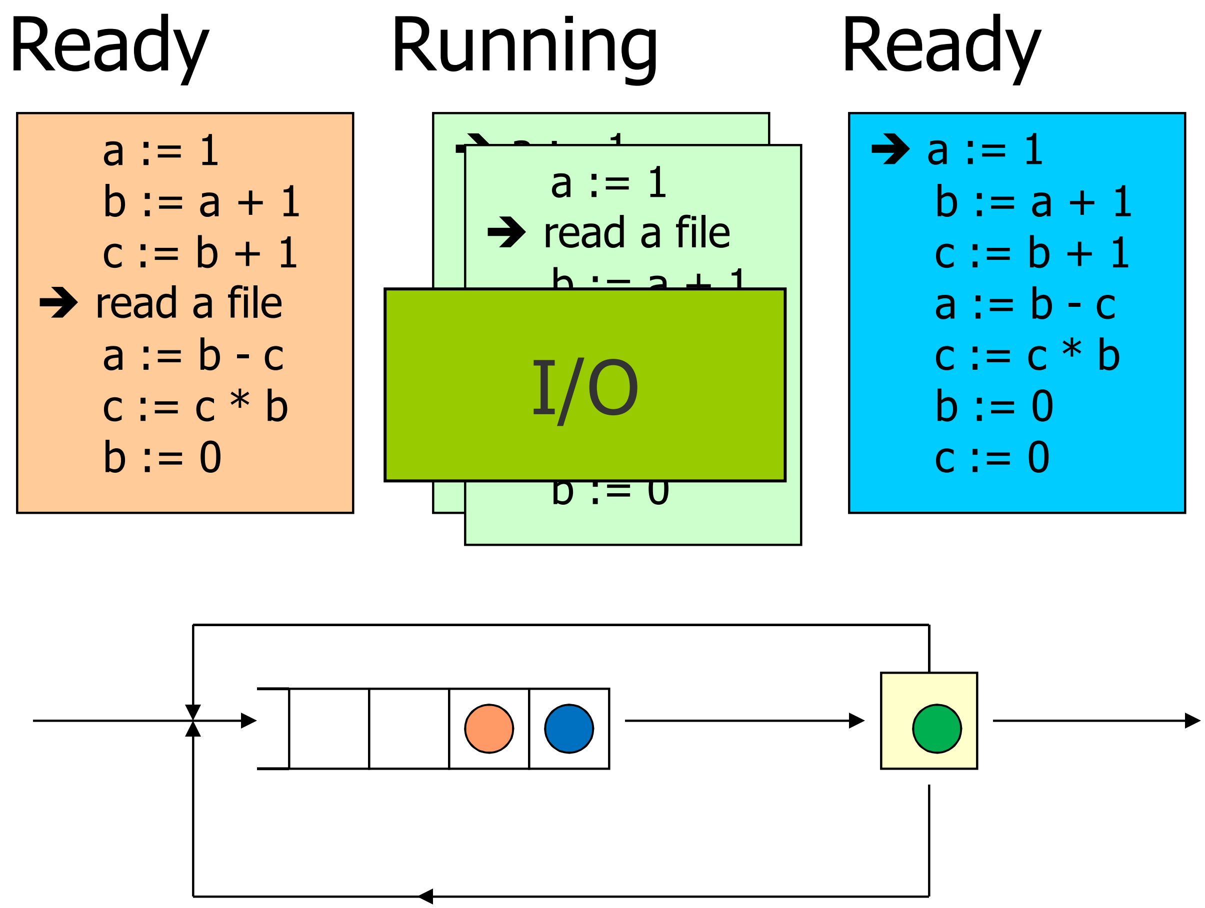
Ready	Ready	Ready
➔ a := 1	➔ a := 1	➔ a := 1
b := a + 1	read a file	b := a + 1
c := b + 1	b := a + 1	c := b + 1
read a file	c := b + 1	a := b - c
a := b - c	a := b - c	c := c * b
c := c * b	c := c * b	b := 0
b := 0	b := 0	c := 0



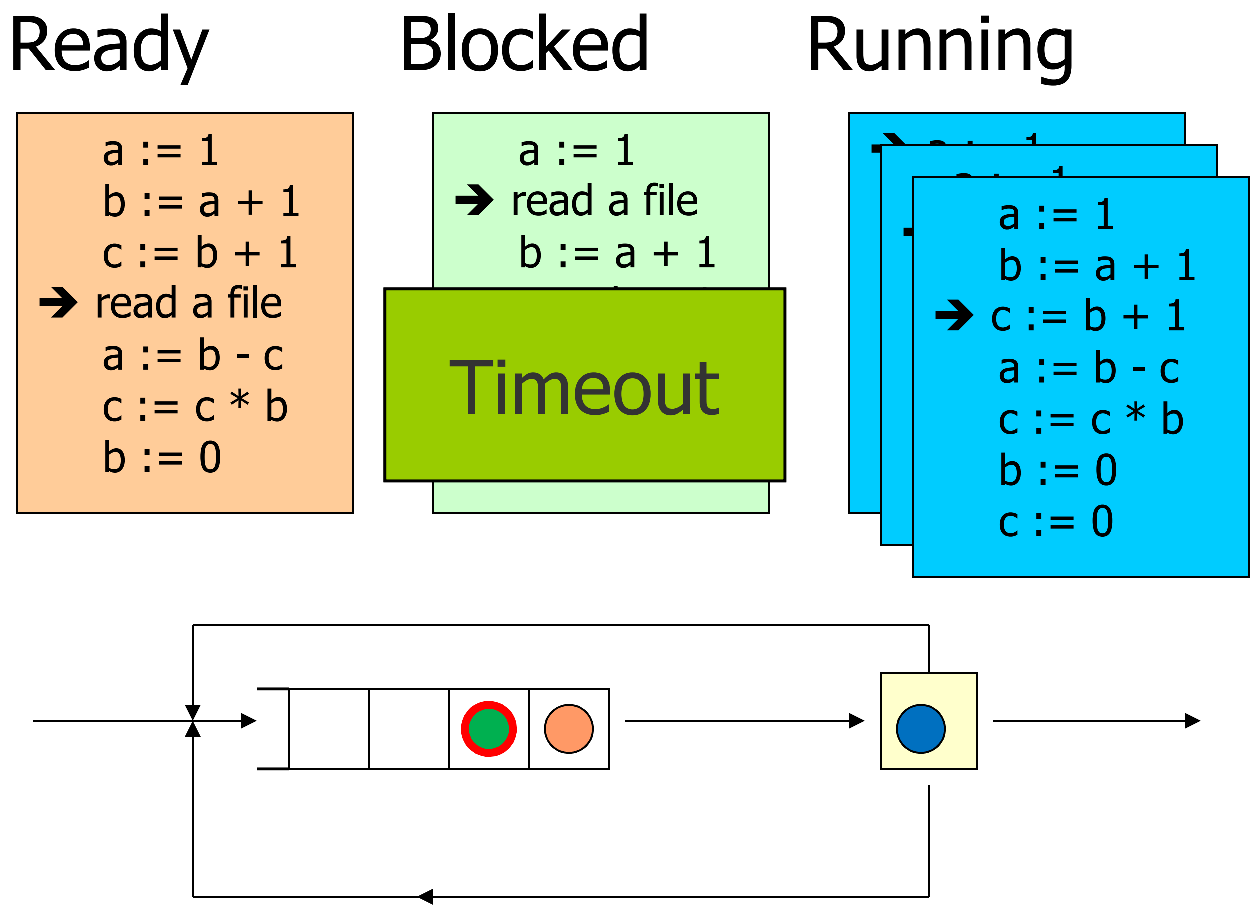
Is Two Process Model Sufficient?



Is Two Process Model Sufficient?

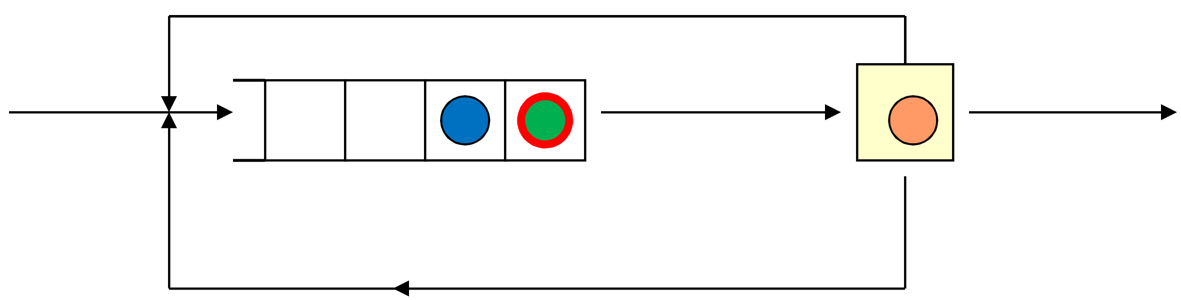


Is Two Process Model Sufficient?



Is Two Process Model Sufficient?

Running	Blocked	Ready
a := 1 b := a + 1 c := b + 1 ➔ read a file a := b - c c := c * b b := 0	a := 1 ➔ read a file b := a + 1 I/O	a := 1 b := a + 1 c := b + 1 ➔ a := b - c c := c * b b := 0 c := 0



Is Two Process Model Sufficient?

Blocked

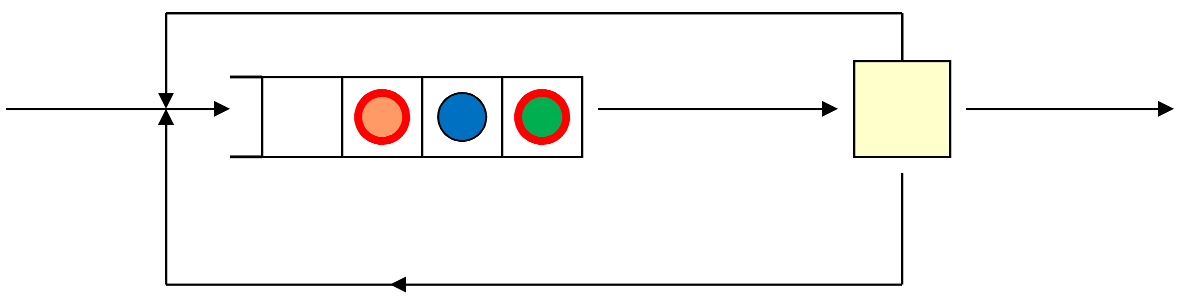
```
  a := 1
  b := a + 1
  c := b + 1
➔ read a file
  a := b - c
  c := c * b
  b := 0
```

Blocked

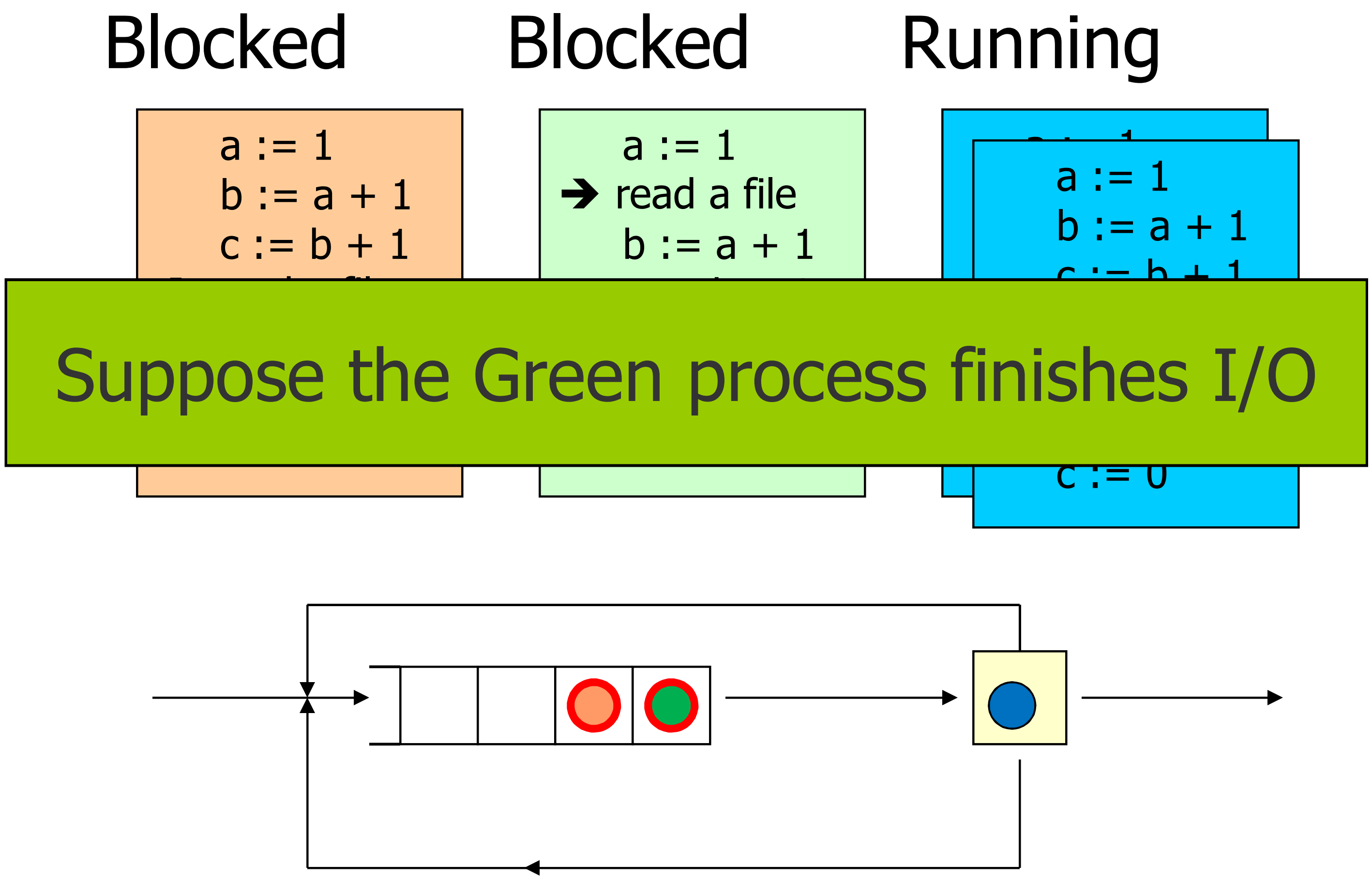
```
  a := 1
➔ read a file
  b := a + 1
  c := b + 1
  a := b - c
  c := c * b
  b := 0
```

```
  b := 0
  c := 0
```

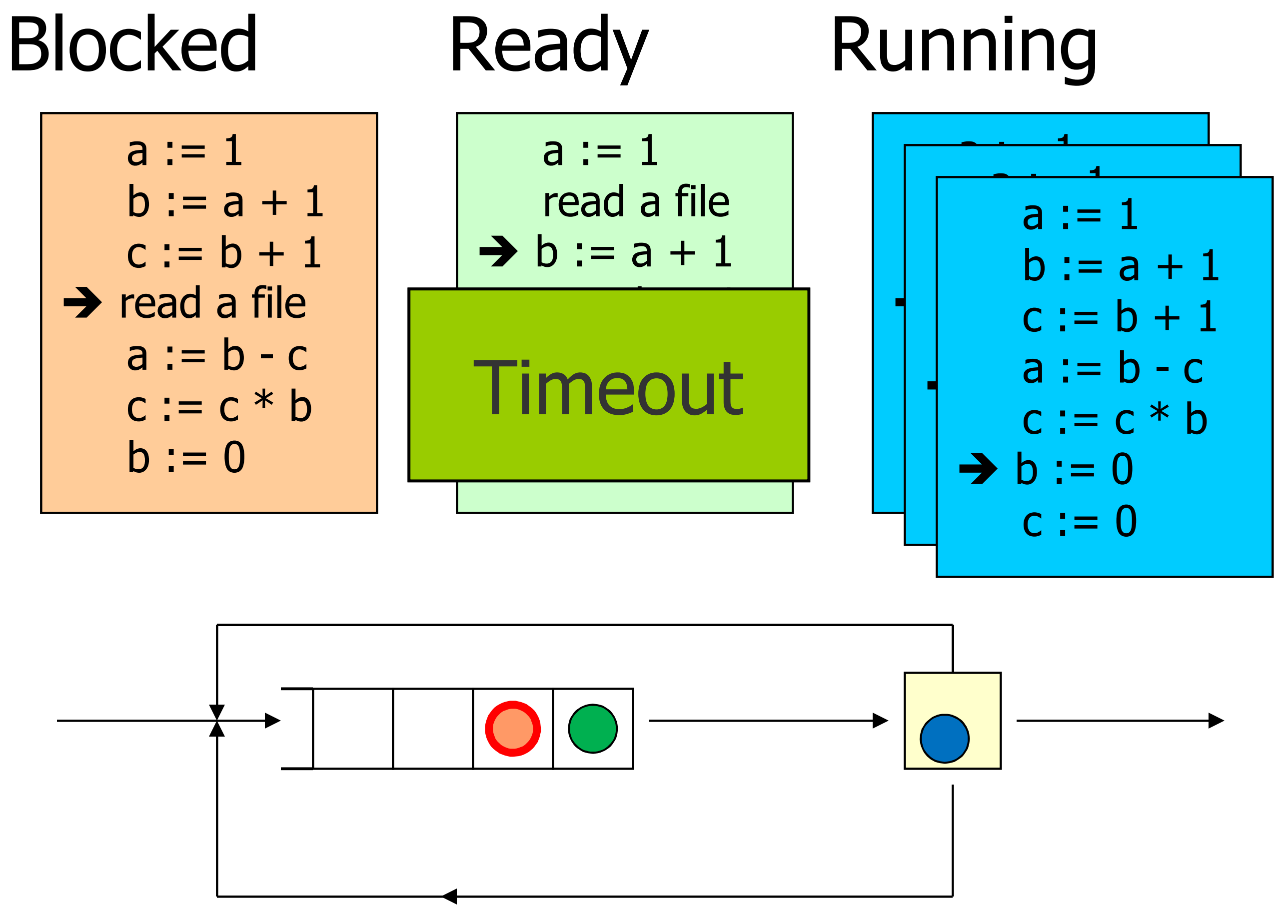
The Next Process to Run cannot be simply selected from the front



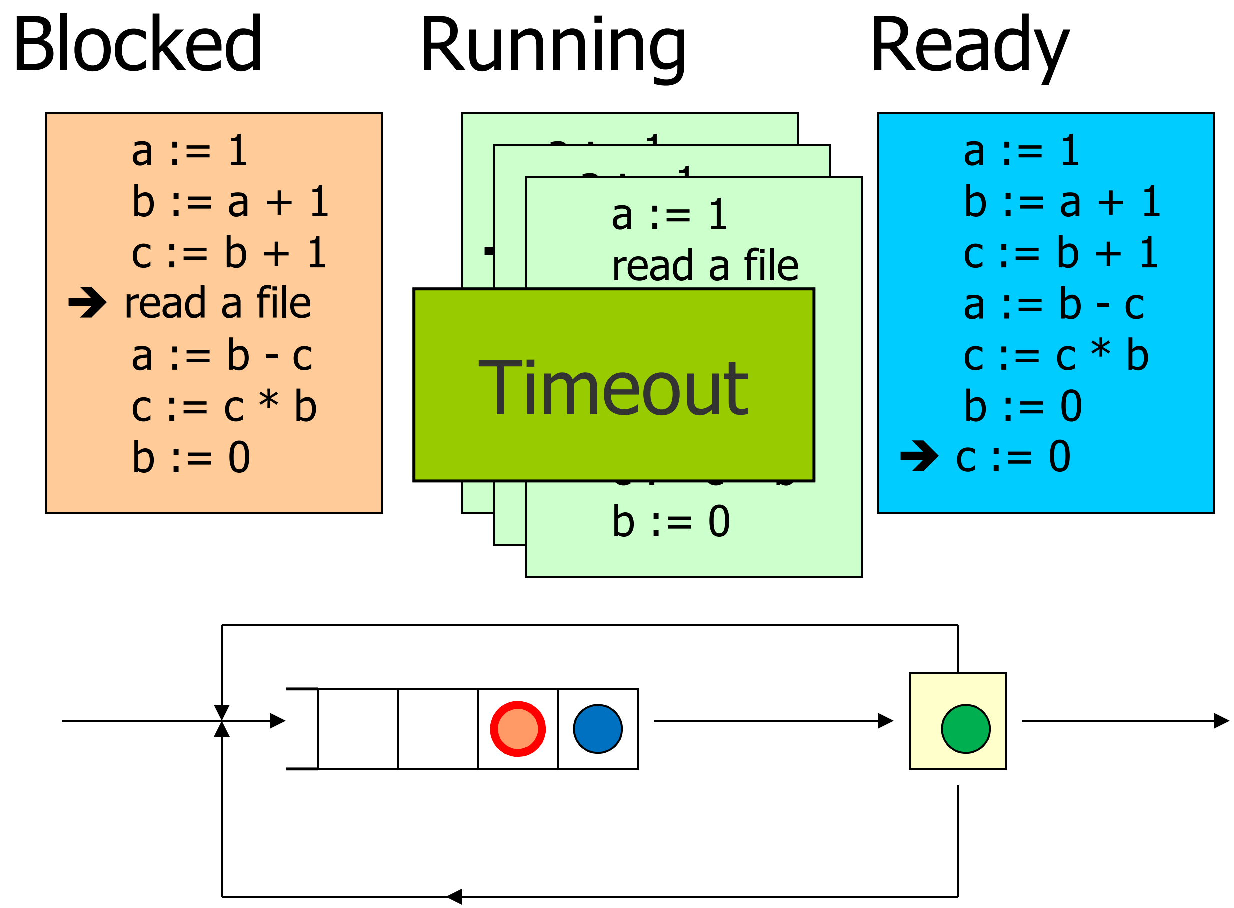
Is Two Process Model Sufficient?



Is Two Process Model Sufficient?



Is Two Process Model Sufficient?



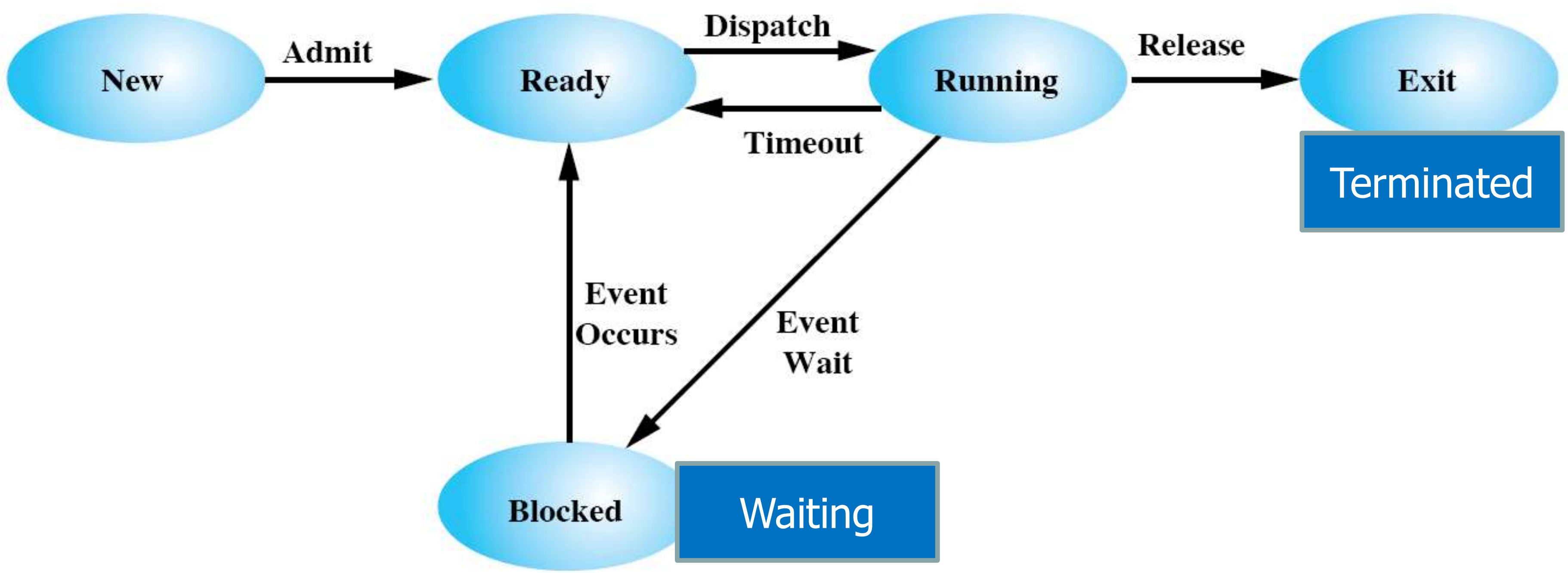
Limitations of Two Process Model

- Timeouts: The process with highest priority will be dispatched to the CPU (… regardless of its capabilities!)

- Wasteful in case of
  - Waiting for I/O operations
  - Waiting for access to a resource
  - In this case dispatcher cannot select the process at the front
  - In worst case dispatcher has to scan the whole queue

- Add more states …
  - Waiting (i.e., blocked), ready

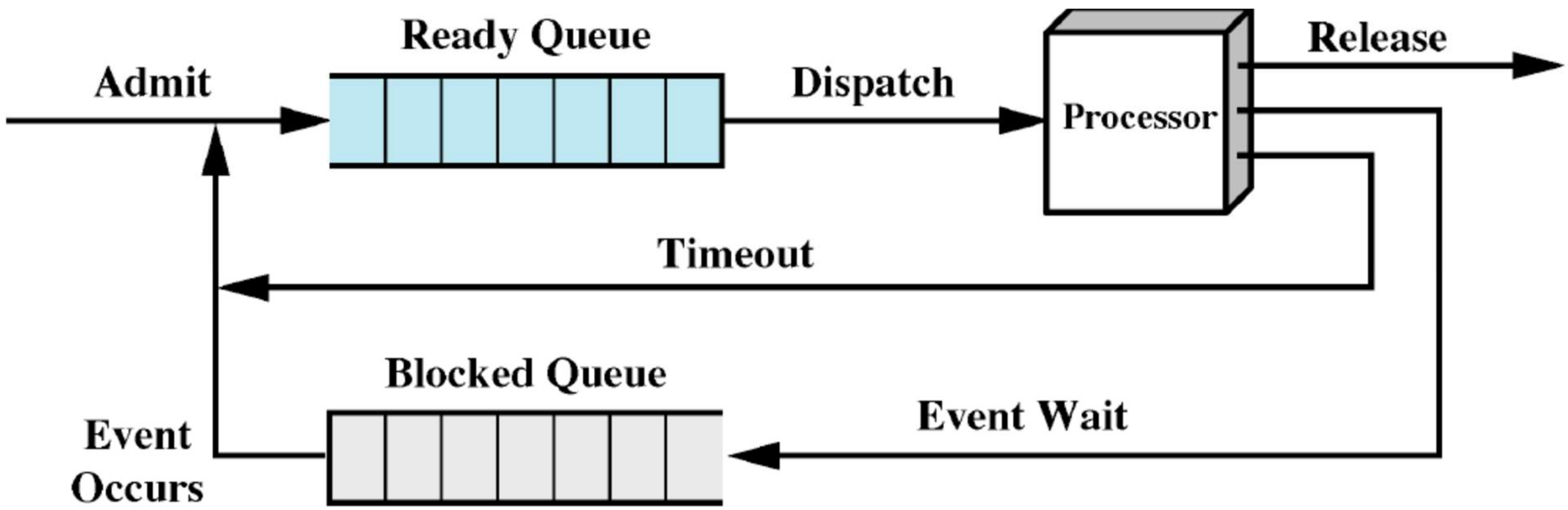
Five State Model



- Admittance to control resource usage
- Processes which need to wait for events can be blocked

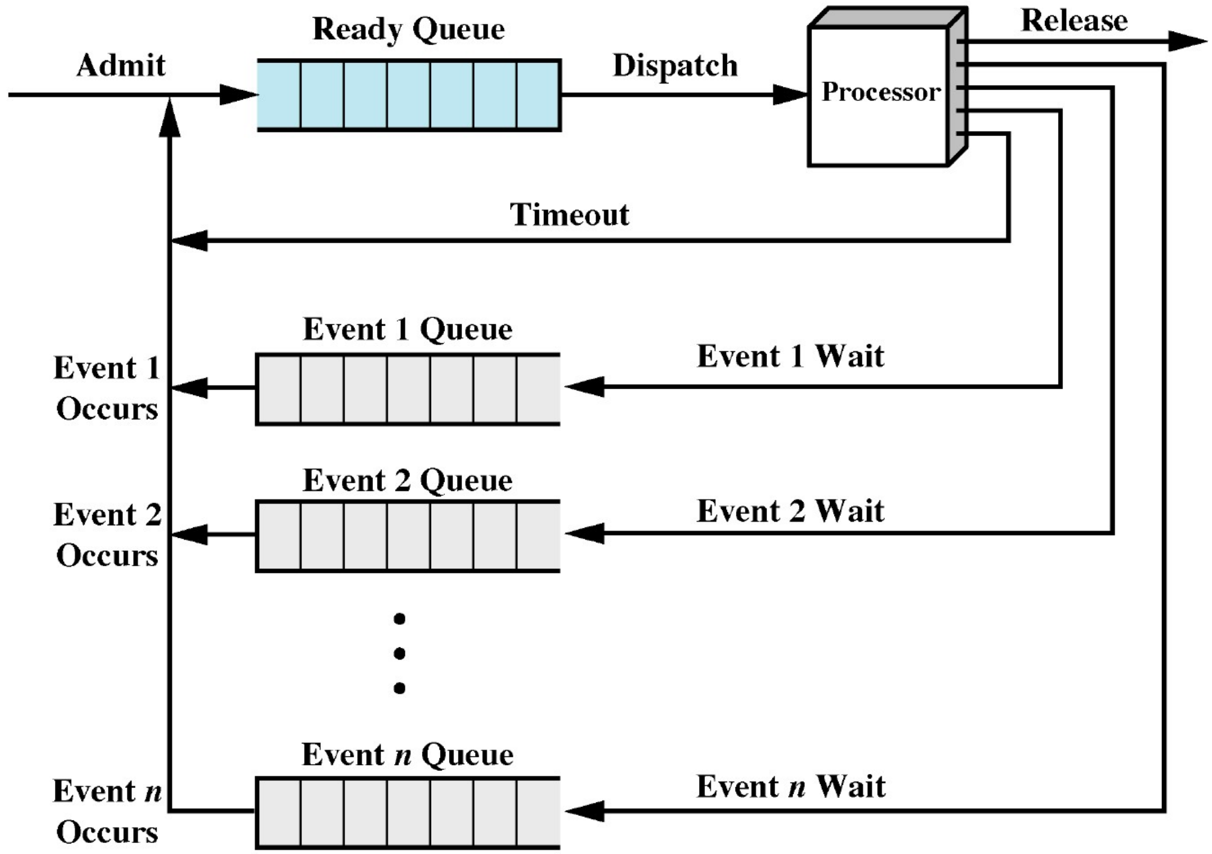
How Can the Dispatcher Use the Five States?

- Maintain different queues
  - One for ready processes
  - One/many for blocked processes
- How expensive is it to transfer a process from blocked to ready state?

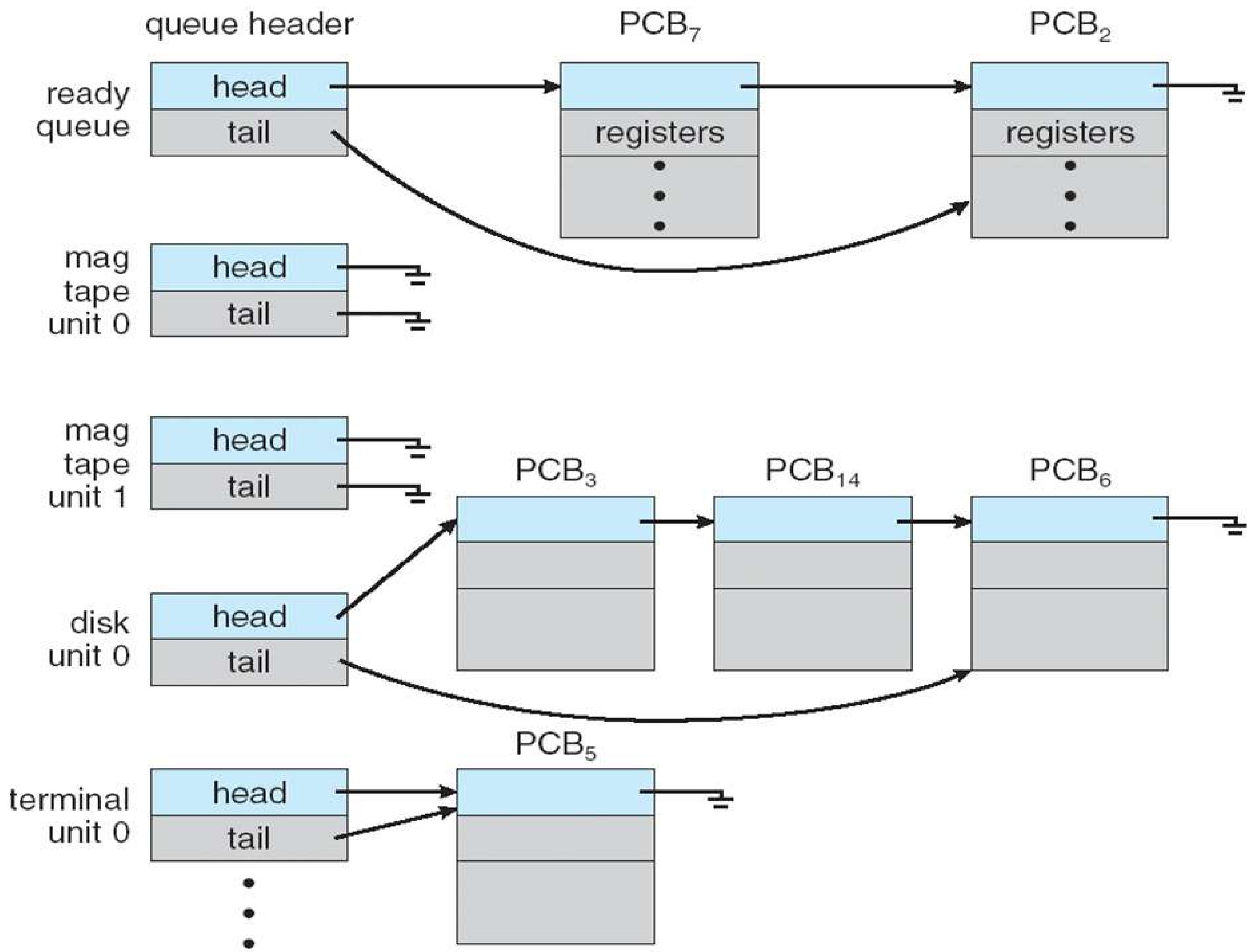


Using Multiple Queues

- Dispatcher can efficiently react to distinct events in the operating system …
- One queue per event
- O(1) operation to transfer process from running to blocked state
- O(1) operation to transfer process from blocked to ready state
- Low dispatch latency



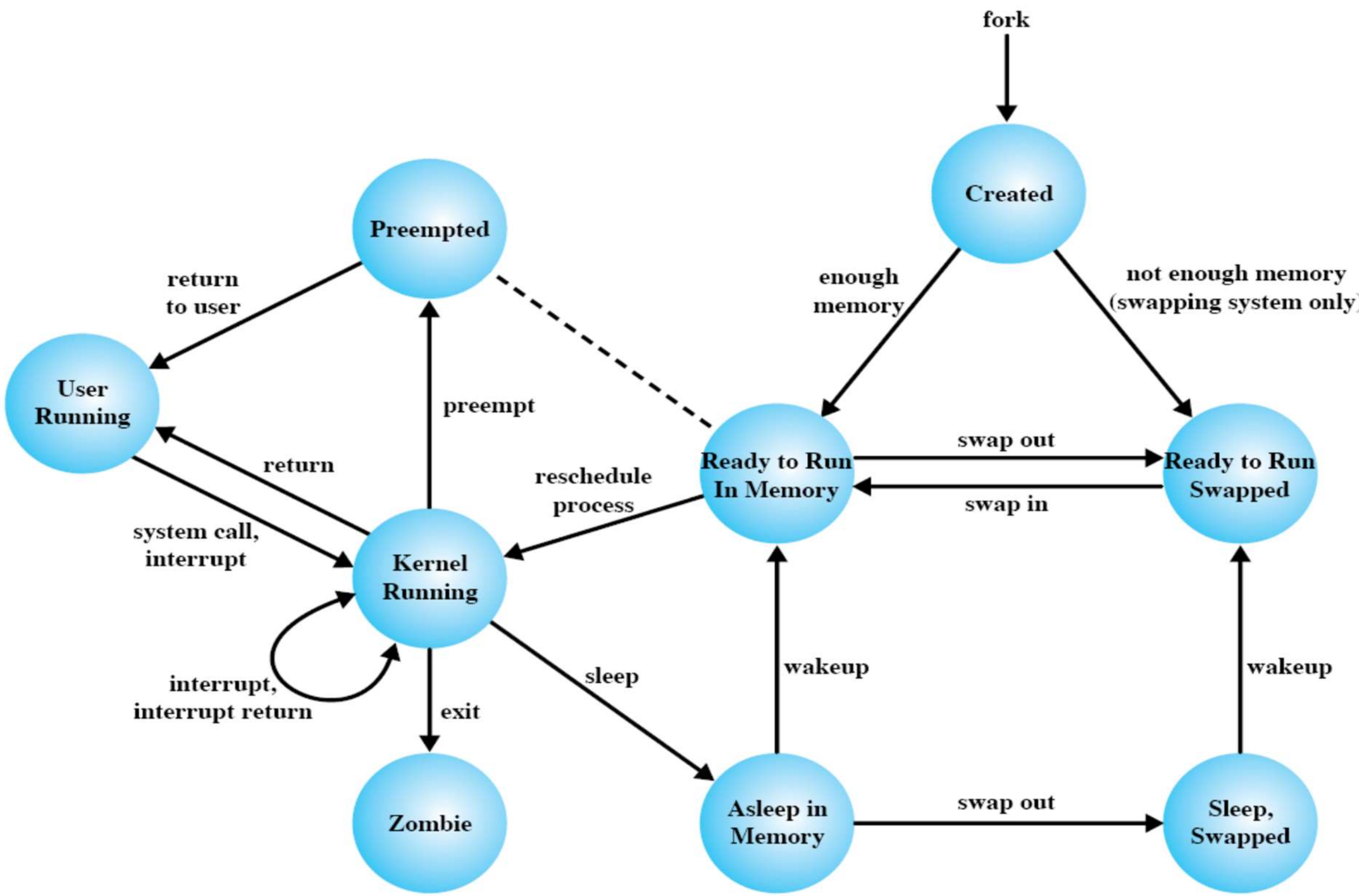
Using Multiple Queues: Ready And I/O Queues



Limitations of the Five State Model

- Available resource usage depends on the number of processes deployed
  - For example, the available memory drops with the number of actively running processes
  - Low hit rate in accessing the memory hierarchy and slow performance
- Idea: Swap some processes to disk
  - For example, free up more memory and achieve higher hit rate for other processes
- Process is in suspend state when swapped to disk
  - Nothing of the process image resides in main memory!

Unix V Process State Transition



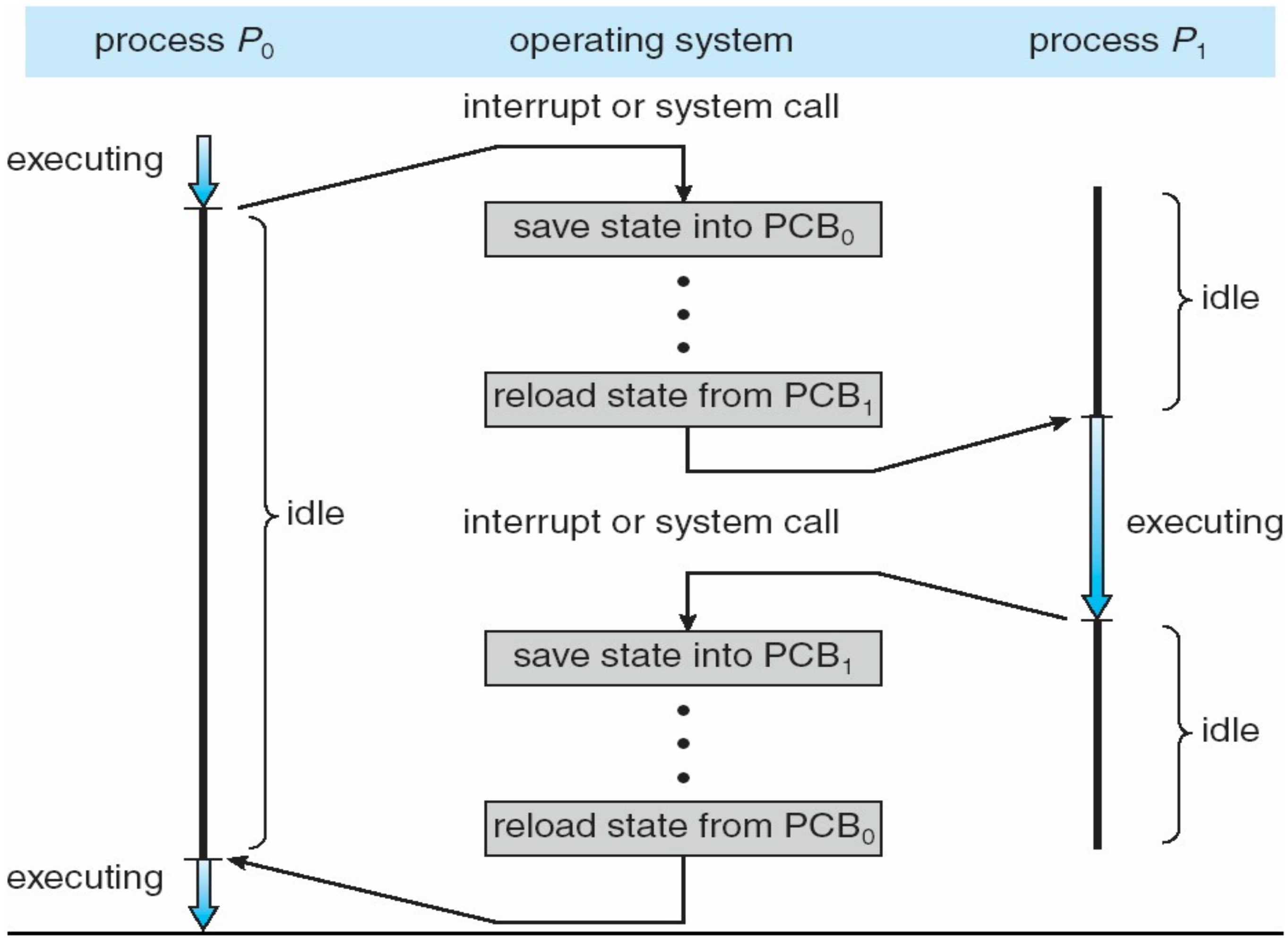
Context Switching

- Context of a process represented in the PCB

- When CPU switches to another process, OS must
  - Save context of processor including program counter and other registers (in PCB)
  - Move PCB to appropriate queue (ready, blocked, …)
  - Select another process for execution
  - Update the PCB of the process selected
  - Update memory-management data structures
  - Restore context (in processor) of the selected process

- Context switch time is overhead
  - The system does no useful work while switching
  - Time is also dependent on hardware support
    - For example, some hardware provides multiple sets of registers per CPU

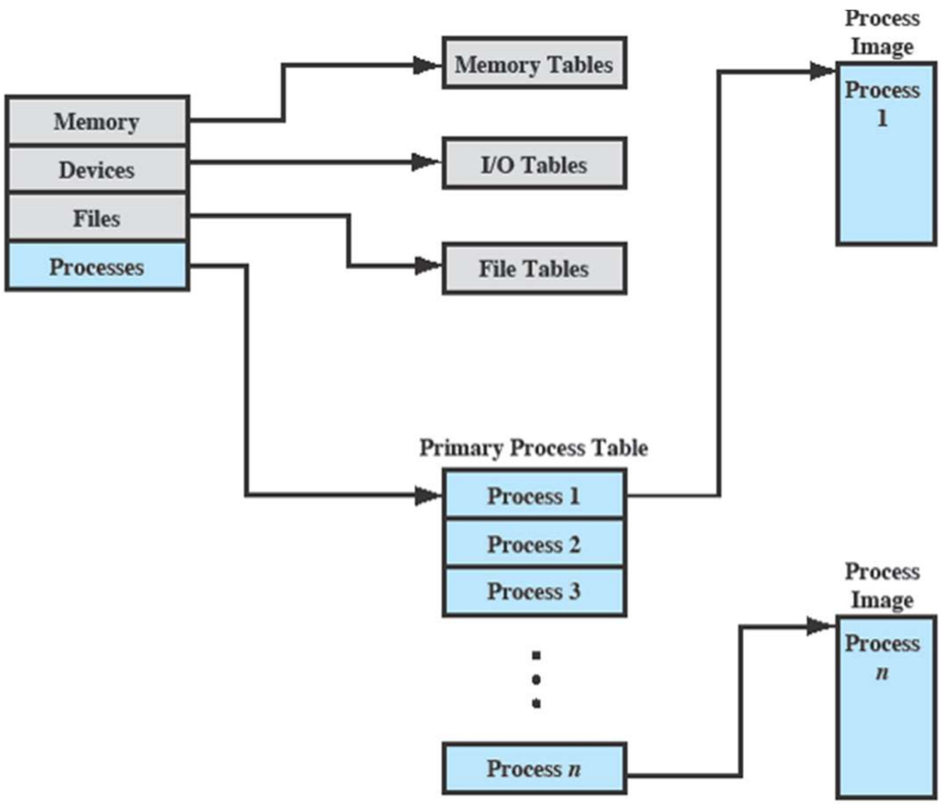
Context Switching

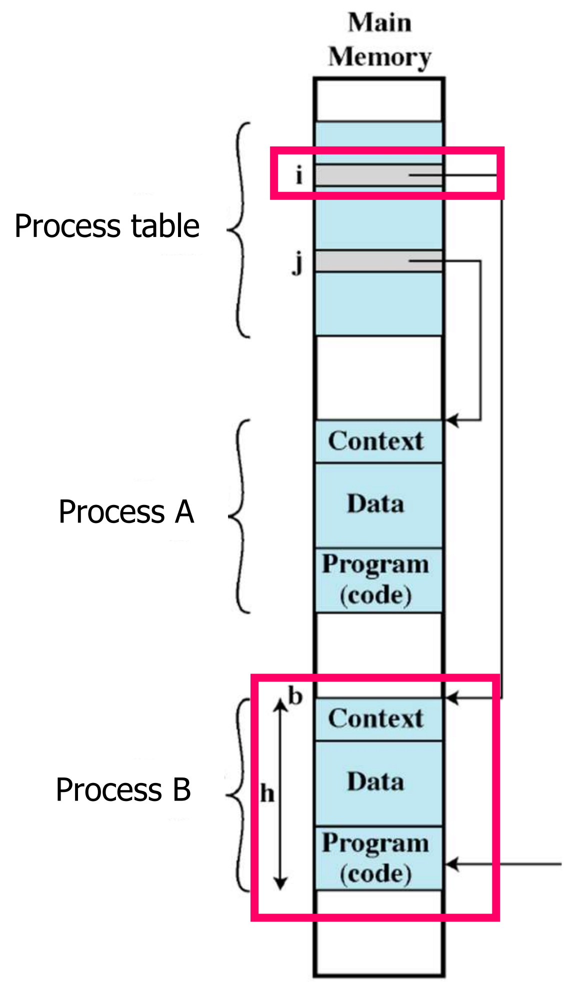


Process Creation

Process Creation

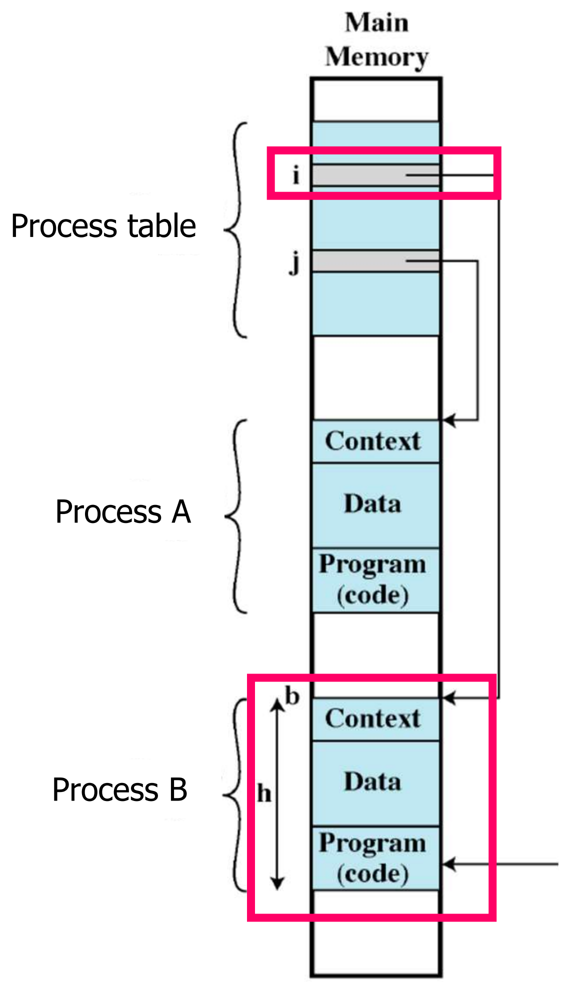
1. Assign a unique identifier (pid) to the new process
2. Allocate space for the process
3. Initialize PCB





Process Creation

1. Assign a unique identifier (pid) to the new process
2. Allocate space for the process
3. Initialize PCB
4. Setup appropriate linkage
   – E.g., add to ready or ready suspend queue
5. Create or expand other data structures



Process Creation

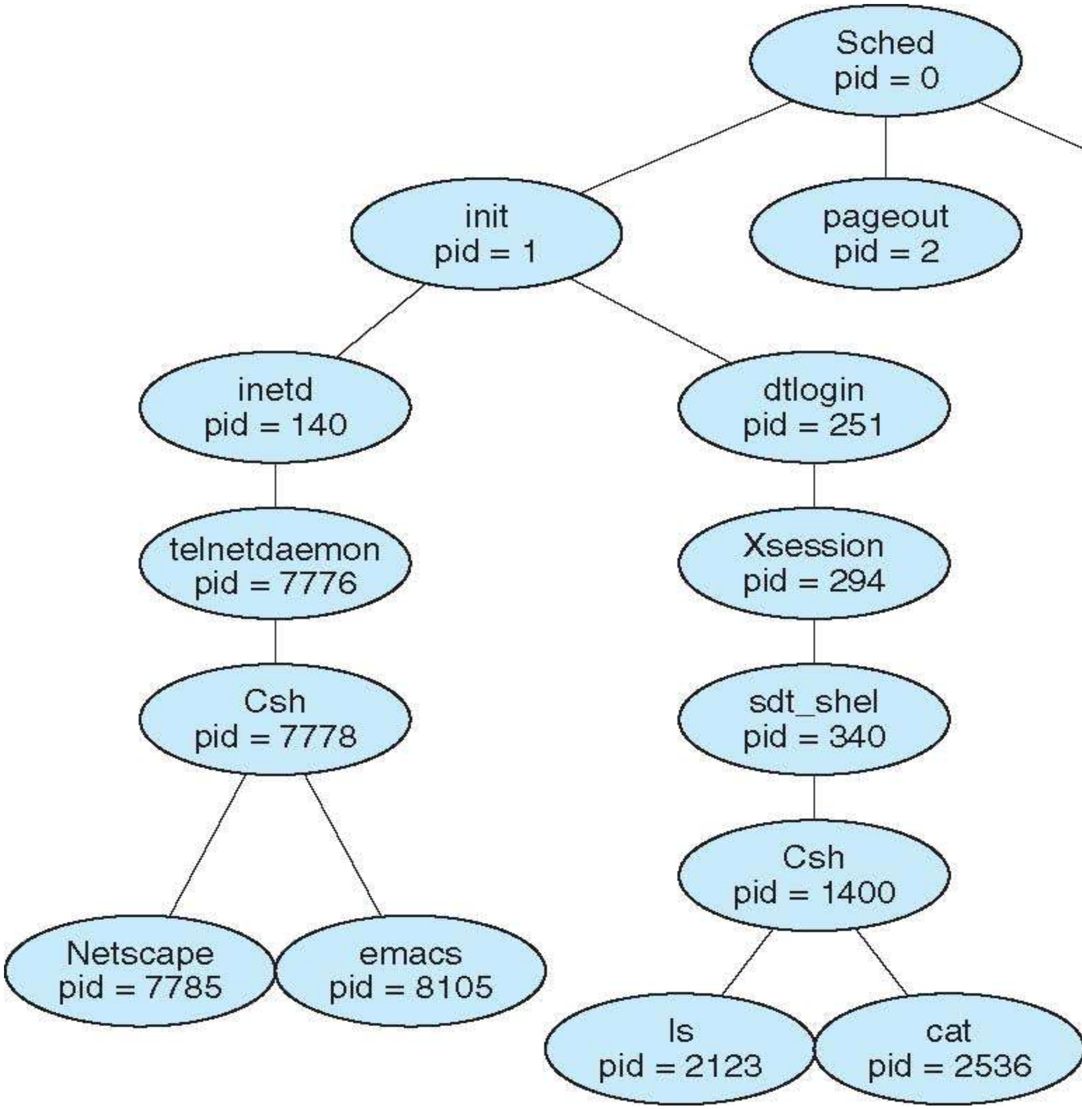
- Traditionally, the OS created all processes
- But it can be useful to let a running process create another
- This action is called process spawning
  - Parent Process is the original, creating, process
  - Child Process is the new process

Process Creation

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

- 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 waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

Process Tree on Solaris



Process Creation in Linux

- **fork** system call creates new process

```
int pid;
int status = 0;

pid = fork();
if (pid>0) {
      /* parent */
      ......
      pid = wait(&status);
} else {
      /* child */
      ......
      exit(status);
}
```

`fork` creates an exact copy of the parent process

process ID to the parent

child pid and status.

`wait` variants allow wait on a specific child, or notification

Child process passes status back to parent on **exit**, to report success/failure

`fork()` System Call

- Child process inherits
  - Stack
  - Memory
  - Environment
  - Open file descriptors
  - Current working directory
  - Resource limits
  - Root directory

- Child process does not inherits
  - Process ID and parent process ID
  - Timers and pending signals
  - Resource utilization and CPU times (initialized to zero)
  - Memory and file locks

Zombie (or Defunct) Process

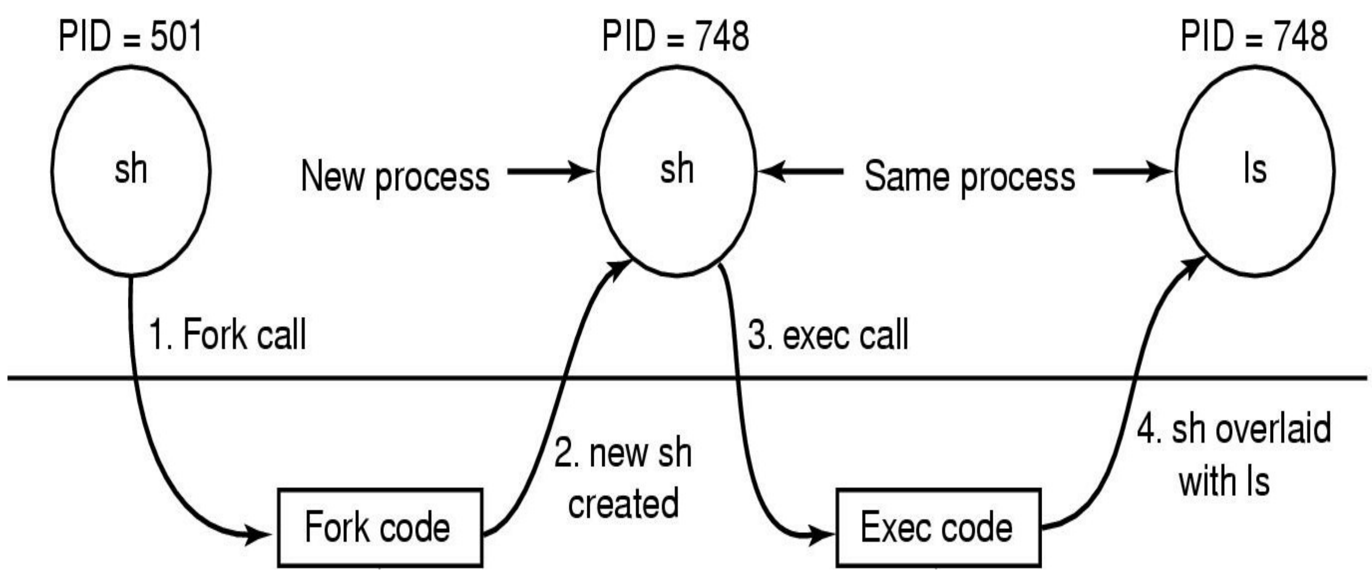
- A process that has completed execution but still has an entry in the process table
- Process table entry is needed to allow the parent process to read the child's exit status
  - Parent can read the status by executing the `wait()` system call
- No memory allocated to zombie process except for the process table entry
  - On exit, all of the memory and resources are deallocated
  - However, process table can only have limited number of entries

Orphan Process

- An orphan process is a process that is still executing, but whose parent has terminated
  - If the parent terminates without calling `wait()`, the child is adopted by `init`
- Orphan processes do not become zombie processes
  - `init` periodically executes the `wait()` system call to avoid zombie processes

`exec()` System Call

- Enable child process to run other program
- Replaces process's memory space with a new program
  - Loads binary file into memory and starts its execution
- Cannot create new process
  - Typically used after `fork()`

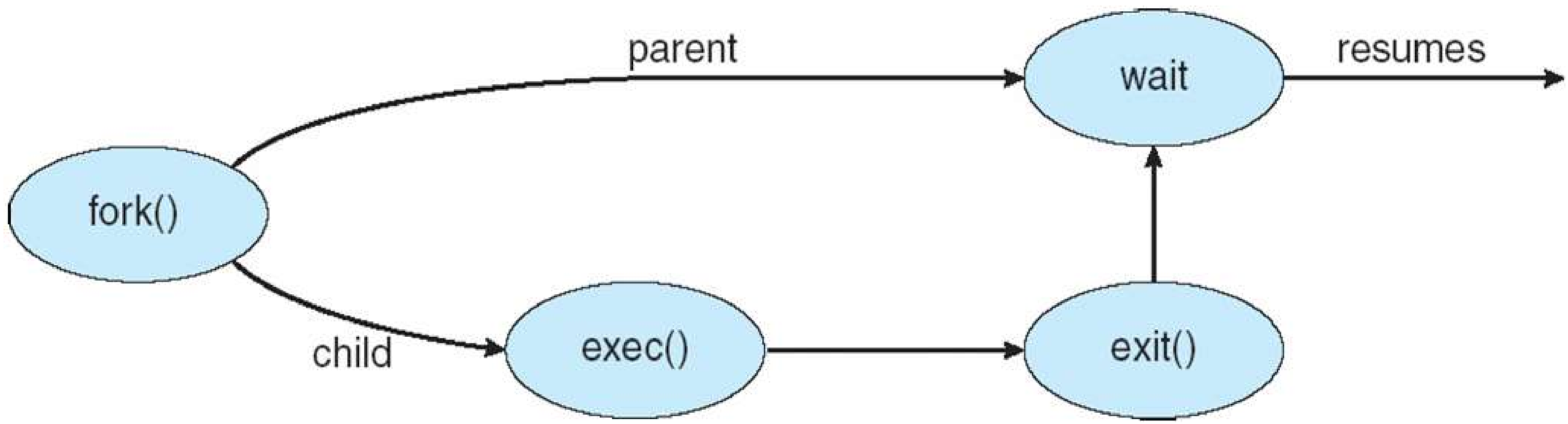


Steps in executing the command **ls** issued to the shell

exec() System Call Example

```
int main(int argc, char *argv[])
{
   pid_t cpid;
   cpid = fork();
   if (cpid == -1){
      perror("fork");
      exit(EXIT_FAILURE);
   }
   if (cpid == 0) {
      /* Child code */
      execlp ("/bin/ls", "ls", NULL);
      /* Why no exit statement*/
   }
   else {
      /* parent code */
      wait(NULL);
      printf("child finished");
      exit(EXIT_SUCCESS);
  }
}
```

`exec()` System Call Example



Interprocess Communication

Interprocess Communication

- A process has access to the memory which constitutes its own address space
- So far, we have discussed communication mechanisms only during process creation/termination
- When a child process is created, the only way to communicate between a parent and a child process is:
  - The parent receives the exit status of the child
- Processes may need to communicate during their life time

Interprocess Communication

- Processes within a system may be independent or cooperating
  - Cooperating process can affect or be affected by other processes

- Reasons for processes corporation
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

- Cooperating processes need interprocess communication (IPC)

- Two fundamental models of IPC
  - Shared memory
  - Message passing

Communication Models

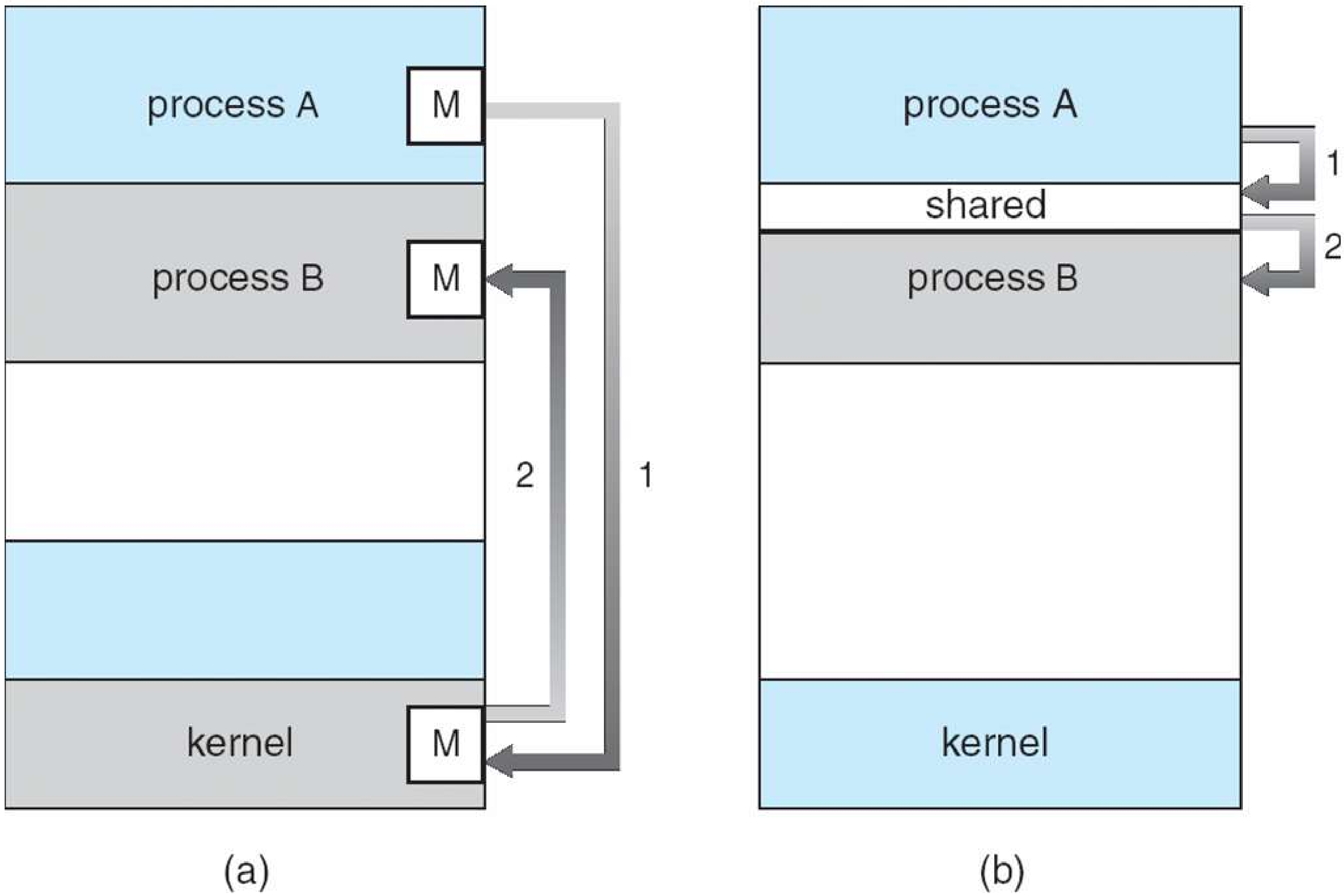
Shared memory

- Need to establish a region of shared memory
- Usually resides in the address space of a process
- OS system calls are only needed to setup the memory
- Basic operations: `read/write`

Message passing

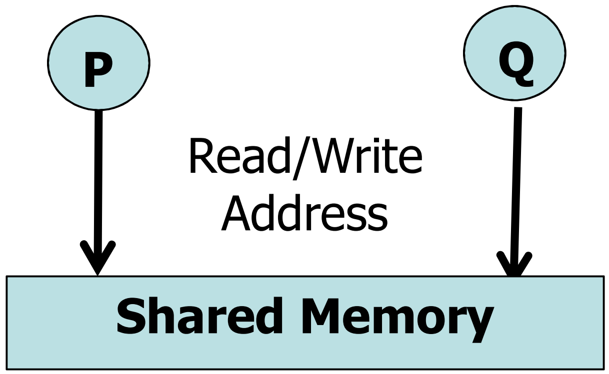
- Send messages between processes
- Requires system calls to the OS for every message
- Often easier to realize for networked process communication
- Basic operations: `send/receive`

Communication Models



Shared Memory

- Usually resides in the address space of the process creating shared memory
- Require processes to coordinate their processing
  - Results of read/write not guaranteed to be deterministic
  - Concurrent writes to the same address
  - Result depends on the order of operations



Shared Memory – POSIX API

- Process first creates shared memory segment
  - `id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);`
- Process wanting access to that shared memory must attach to it
  - `shared_memory = (char *) shmat(id, NULL, 0);`
- Now the process could write to the shared memory
  - `sprintf(shared_memory, "Writing to shared memory");`
- When done a process can detach the shared memory from its address space
  - `shmdt(shared_memory);`

Message Passing

- No sharing of resources between processes needed
- Basic operations
  - `send({destination}, message)`: send a message
  - `receive({source}, message)`: receive a message
- If 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)

Implementation Questions

- 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?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

Direct symmetric communication

- Each process that wants to communicate must name the recipient
  - `send(P, message)`
    - ➢ Send a message to process P
  - `receive(Q, message)`
    - ➢ Receive a message from process Q
- Communication link established automatically
- Link associated exactly between two processes

Direct asymmetric communication

- Use instead
  - `send(P, message)`
    - ➢ Send a message to process P
  - `receive(message)`
    - ➢ Receive message from arbitrary process

Indirect Communication

- Problem of Direct Addressing
  - Changing a process identifier may impact all other process definitions
  - Limited modularity

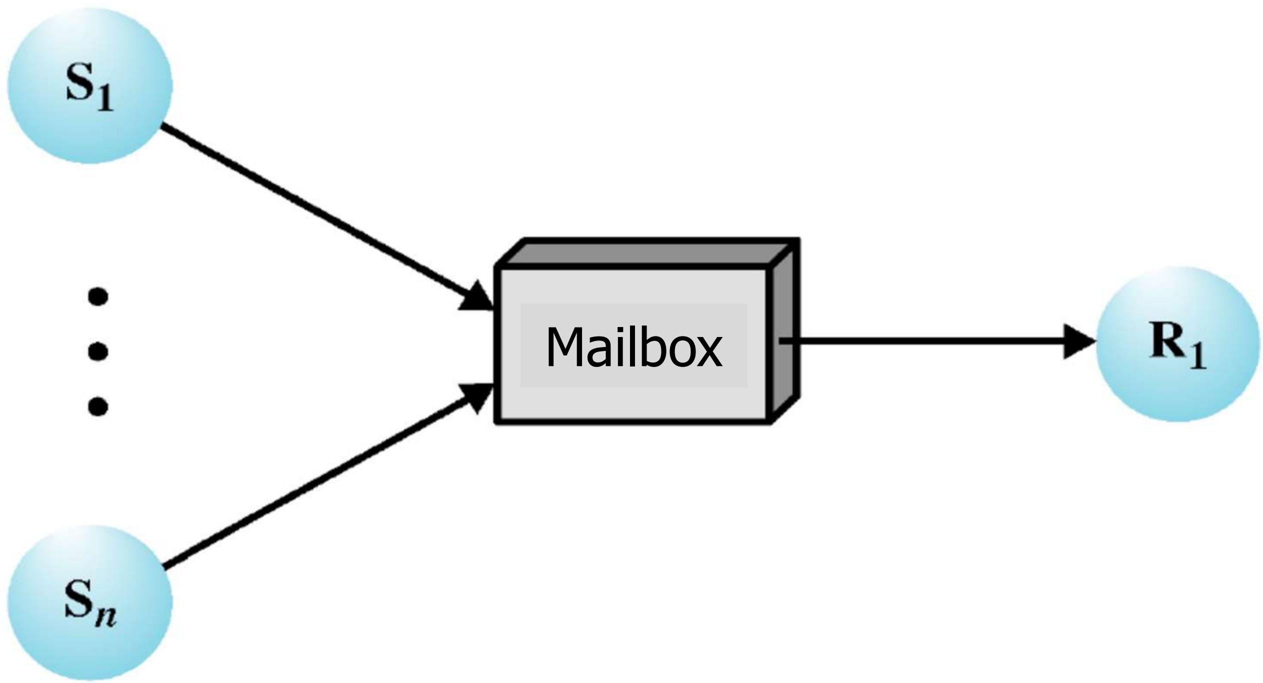
- Indirect addressing uses mailbox or port concept
  - Mailbox: Shared by multiple receiver
  - Port: Receiver Specific, e.g., WebServer uses port 80
  - `send(A, message)`
    - Send a message to mailbox/port named A
  - `receive(A, message)`
    - Receive a message from mailbox A
  - Link is only established if both members share a mailbox
  - A link may be associated with more than 2 processes
  - Each pair of processes may share several communication links

One-to-One

- Private communication link between two processes

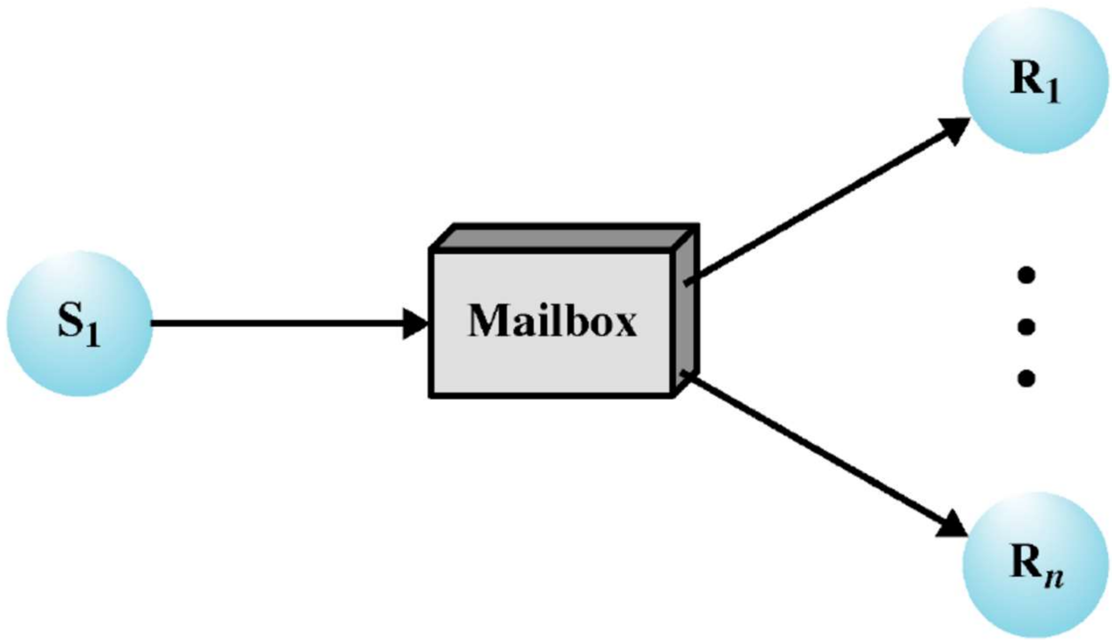
Many-to-One

- Client/server interactions



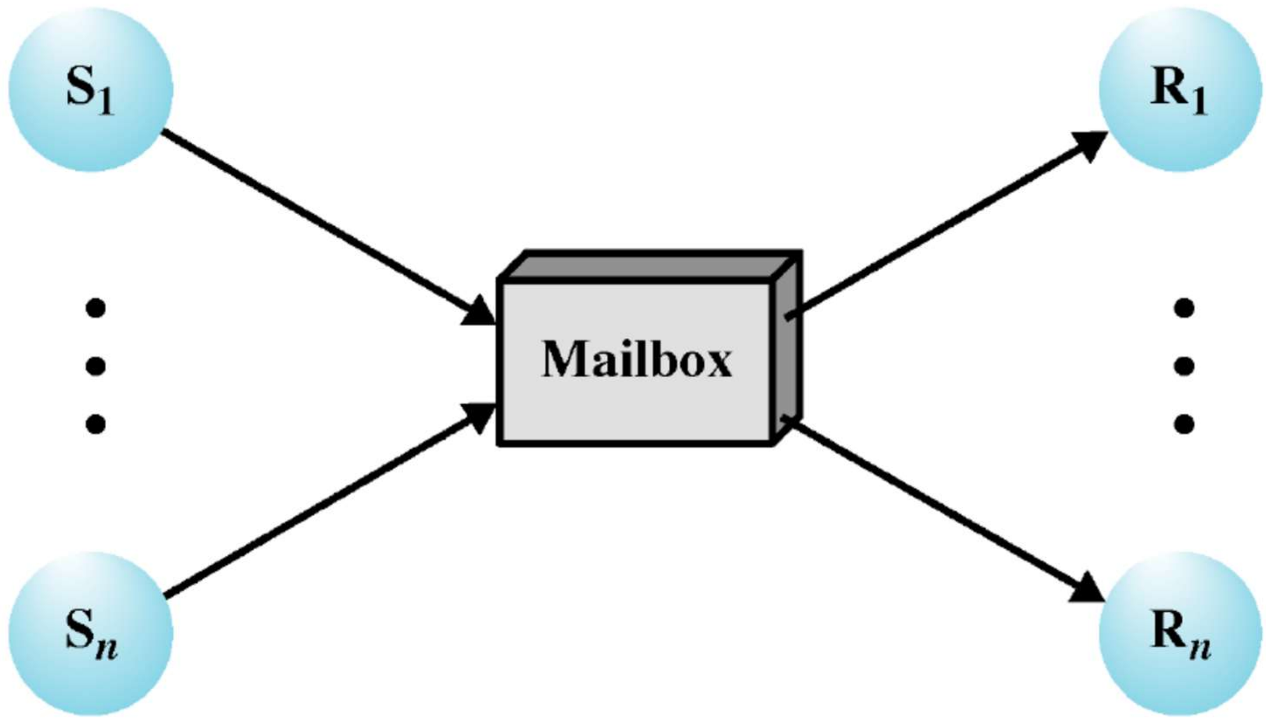
One-to-Many

- Multicast information from a source to a set of receivers



Many-to-Many

- Allows for many-to-many communication



Indirect Communication – Example

- Mailbox sharing
  - P1, P2, and P3 share mailbox A
  - P1, sends; P2 and P3 receive
  - Who gets the message?

- Possible 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 of the receiver

Indirect Communication – Ownership

- Could be either process or Operating System

- Case process is the owner of a mailbox
  - Owner performs as the receiver of messages
  - User sends messages to the mailbox
    - Mailbox destroyed with termination of the process

- Case OS is the owner
  - Need support for processes to
    - Create
    - Send and receive messages through the mailbox
    - Delete a mailbox
  - Manage ownership
  - Possibly multiple owners/ receivers

Indirect Communication – Ownership

- Process owns (i.e. mailbox is implemented in user space)
  - Only the owner may receive messages through this mailbox
  - Other processes may only send
  - When process terminates any "owned" mailboxes are destroyed

- Kernel owns
  - Kernel provides mechanisms to create, delete, send and receive through mailboxes
  - Mailbox has existence of its own independent of any process
  - Process that creates mailbox owns it (and so may receive through it)
  - Process may transfer ownership to another process

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
- Combinations of blocking and non blocking calls
  - Blocking send + blocking receive
    - Tight coupling between processes
  - Non blocking send + blocking receive

Buffering

- All messaging system require framework to temporarily buffer messages (i.e., Queues)

- Zero capacity
  - No messages may be queued within the link
  - Requires sender to block until receiver retrieves message

- Bounded capacity
  - Link has finite buffer capacity to hold messages
  - If link is full then sender must block until one is freed up

- Unbounded capacity
  - Link has unlimited buffer space
  - Send never needs to block

IPC Case Study: Unix Pipes

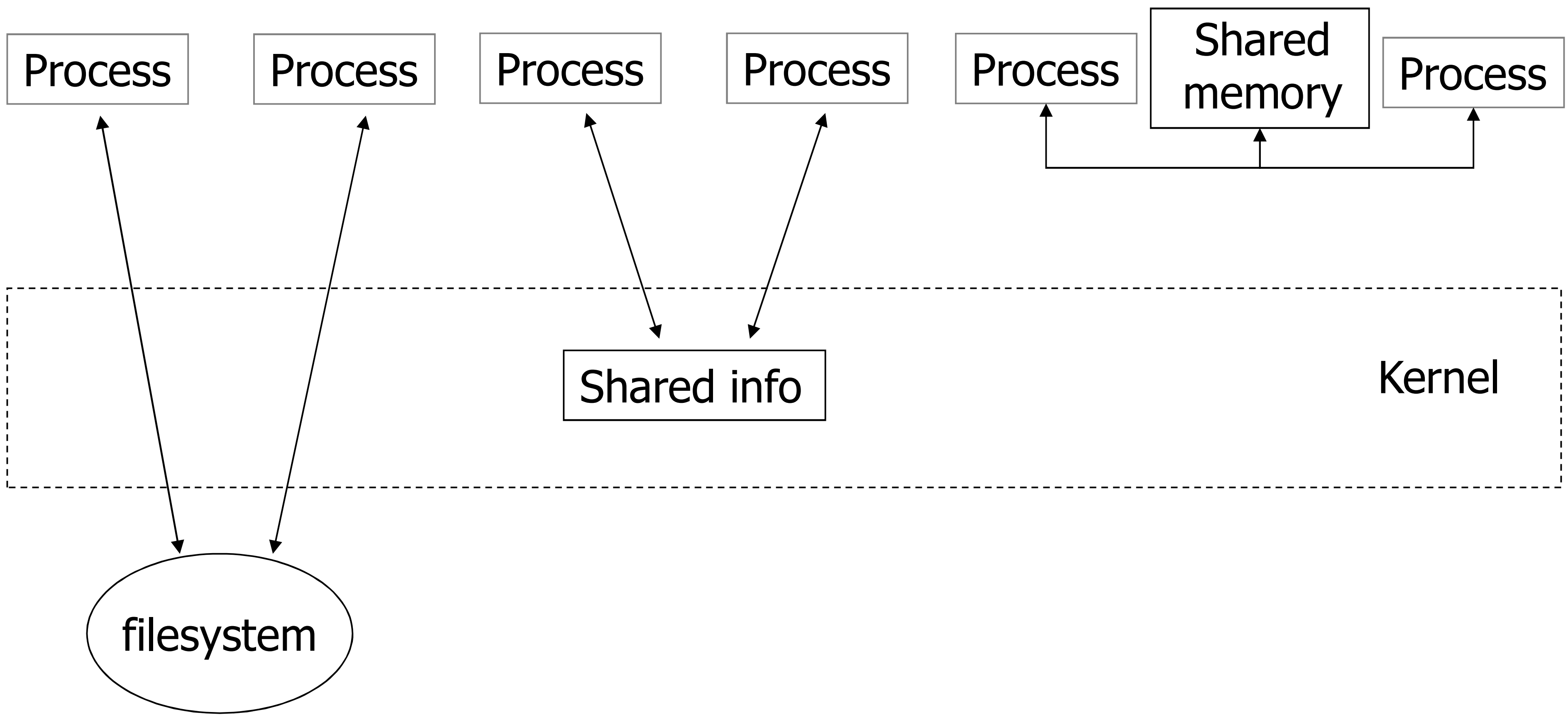
Process Creation Recap

- Consider the following code

```
for(int i = 0; i < 4; i++){
    fork();
}
```

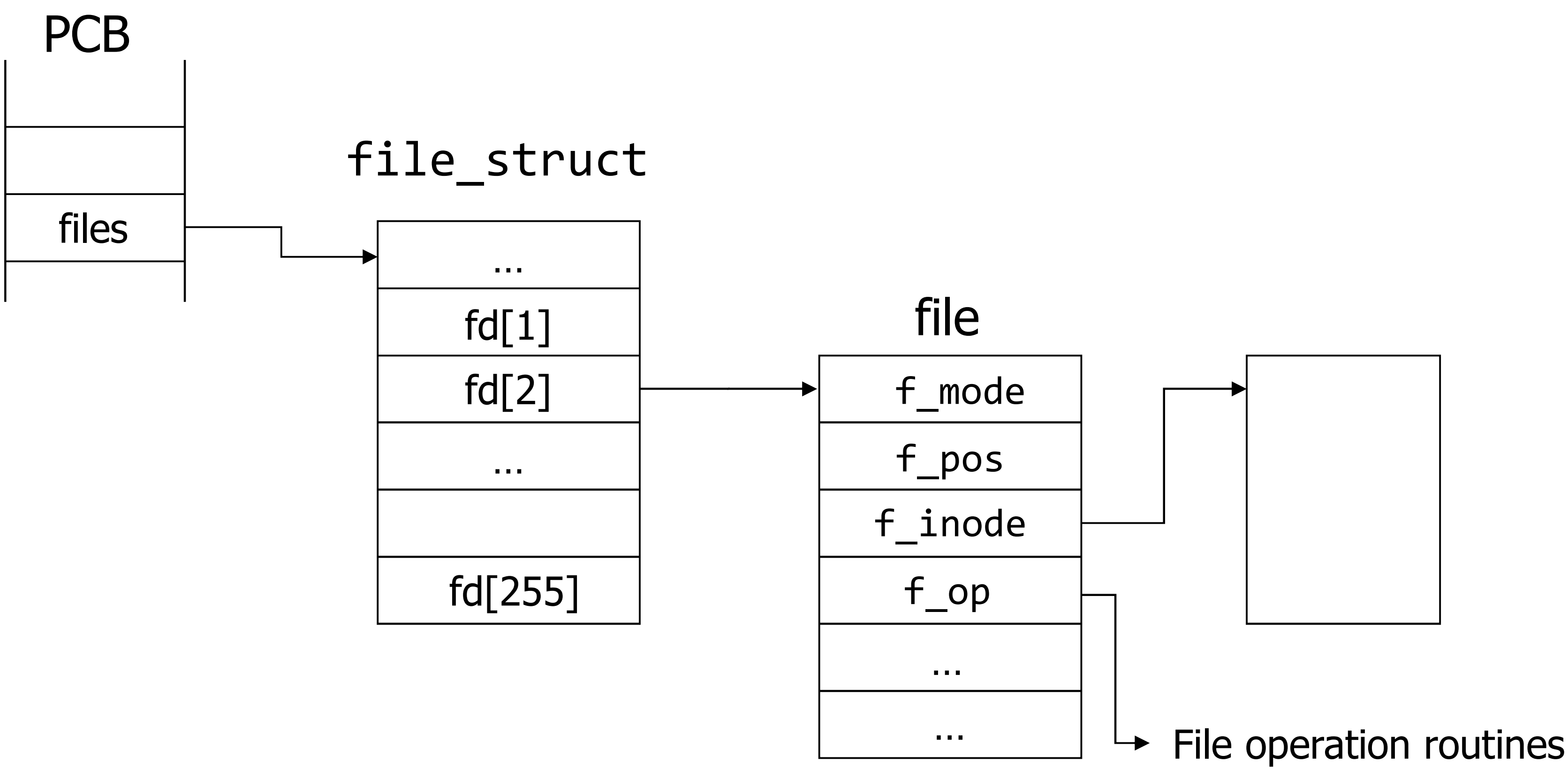
- How many process would be created?
- Represent the processes in form of tree

Unix IPC


Process
Process
Process
Process
Process
Shared memory
Process
Shared info
Kernel
filesystem

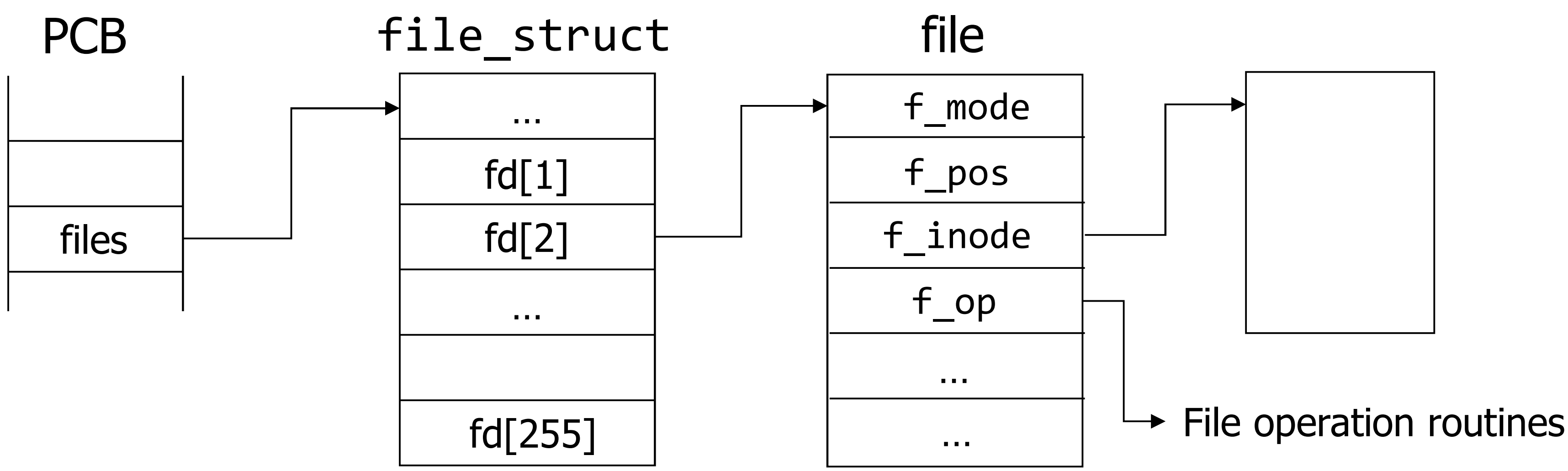
File Descriptors

- PCB of each process keeps track of open files
- Pointer to `file_struct`, a kernel-resident array data structure containing the details of open files



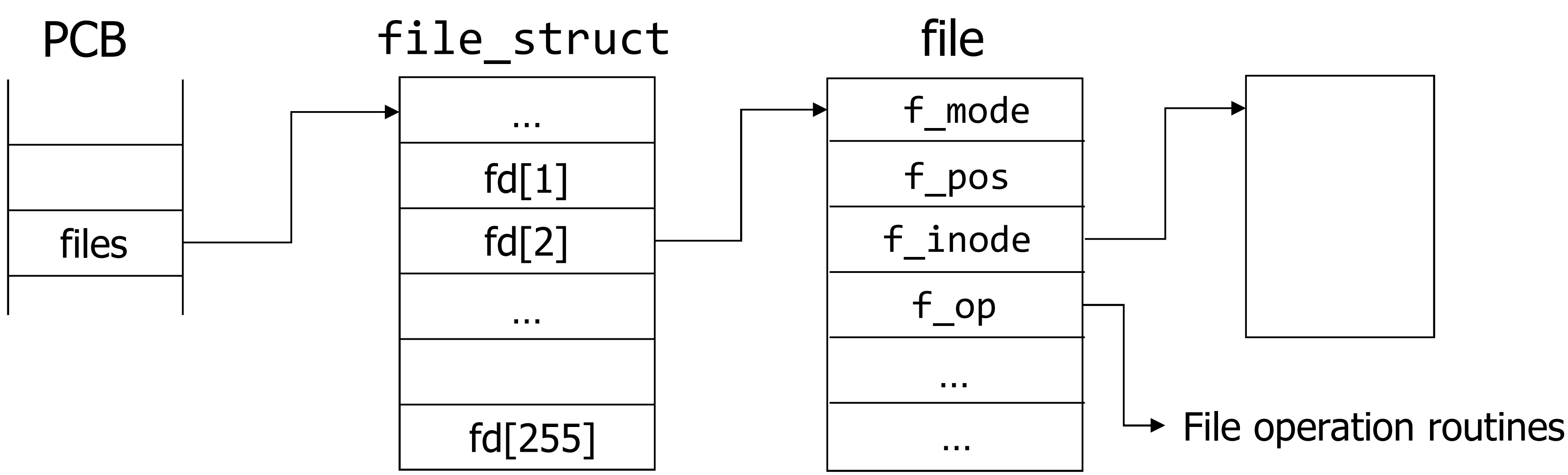
File Descriptors

- `files_struct` contains pointers to file data structures
  - Each one describes a file being used by this process
- `f_mode`: Describes file mode, read only, read and write or write only
- `f_pos`: Holds the position in the file where the next read or write operation will occur
- `f_inode`: Points at the actual file



File Descriptors

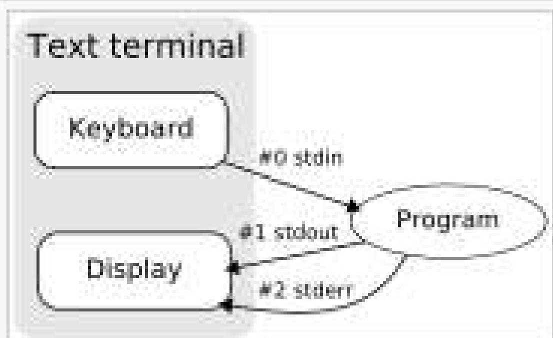
- When a process opens a file, one of the free file pointers in the `files_struct` is used to point to the new file structure
- When Unix processes do any sort of I/O, they do it by reading or writing to a file descriptor
  - A file descriptor is simply an integer associated with an open file
- All accesses to files are via standard system calls which pass or return file descriptors



Standard Input, Output, Error

- Linux processes expect three file descriptors to be open when they start
  - Standard input: File descriptor 0 (`stdin`)
  - Standard output: File descriptor 1 (`stdout`)
  - Standard error: File descriptor 2 (`stderror`)

- These three are usually inherited from the creating parent process



The file descriptors for input, output, and error

Standard Input, Output, Error: Example

- Read from standard input (by default it is keyboard)
  - `char buffer[10];`
  - `read(0,buffer,5);`

- Write to standard output (by default is is monitor))
  - `char buffer[10];`
  - `write(1,buffer,5);`

- By changing the file descriptors we can write to files

- `fread/fwrite` etc. are wrappers around the above `read/write` functions

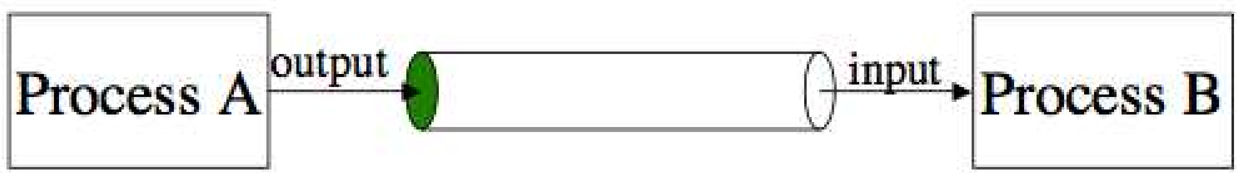
Unix Fact

- Everything in Unix is a file

- A file in Unix can be
  - A network connection
  - A FIFO queue
  - A pipe
  - A terminal
  - A real on-the-disk file
  - Or just about anything else

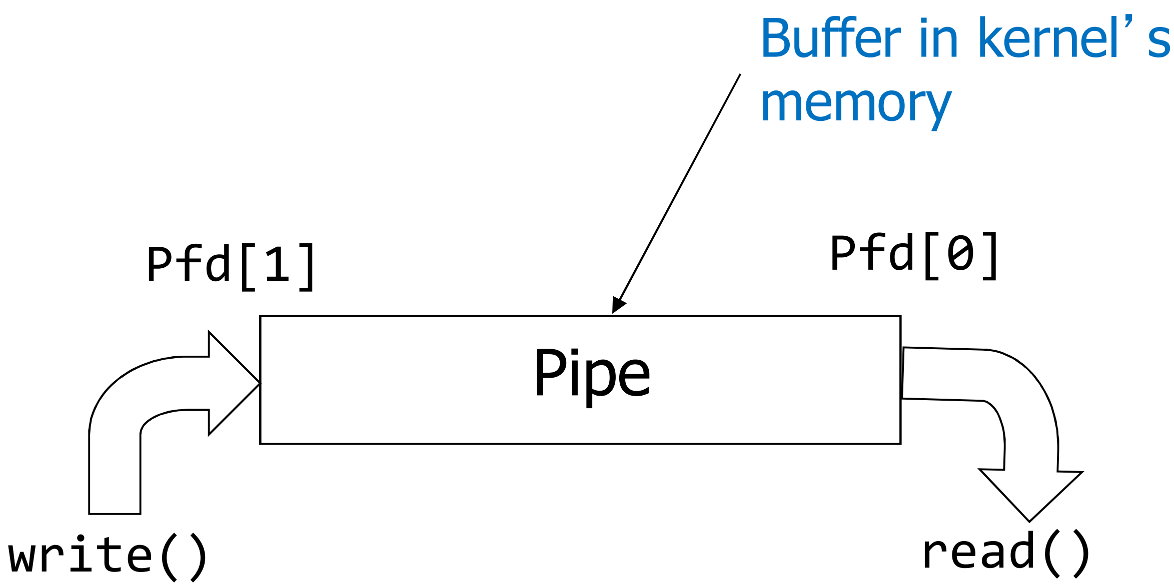
Pipes

- Pipes represent a channel for Interprocess Communication
  - Provides a one-way flow of data
  - Can be thought as a special file that can store a limited amount of data in a first-in-first-out manner, exactly akin to a queue



Pipes

- Shared info in kernel’s memory



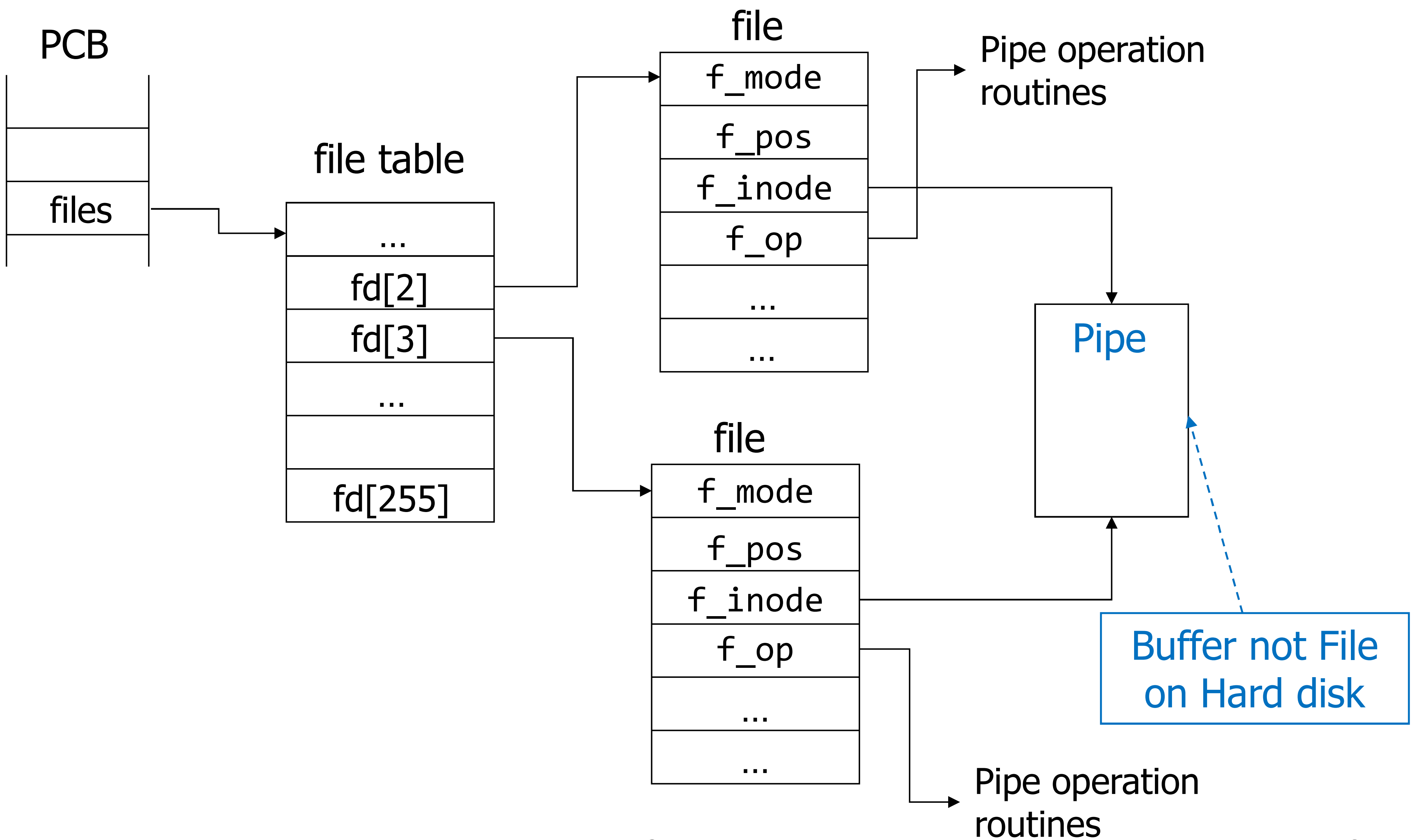
Pipes

- A pipe is implemented using two file data structures which both point at the same temporary data node

- This hides the underlying differences from the generic system calls which read and write to ordinary files

- Thus, reading/writing to a pipe is similar to reading/writing to a file

Pipes


PCB
files
file table
...
fd[2]
fd[3]
...
fd[255]
file
f_mode
f_pos
f_inode
f_op
...
...
file
f_mode
f_pos
f_inode
f_op
...
...
Pipe operation routines
Pipe operation routines
Pipe
Buffer not File on Hard disk

(Unnamed) Pipe Creation

Two methods for creating (unnamed) pipes

- pipe system call

```
#include <unistd.h>
int pipe(int filedes[2]);
```

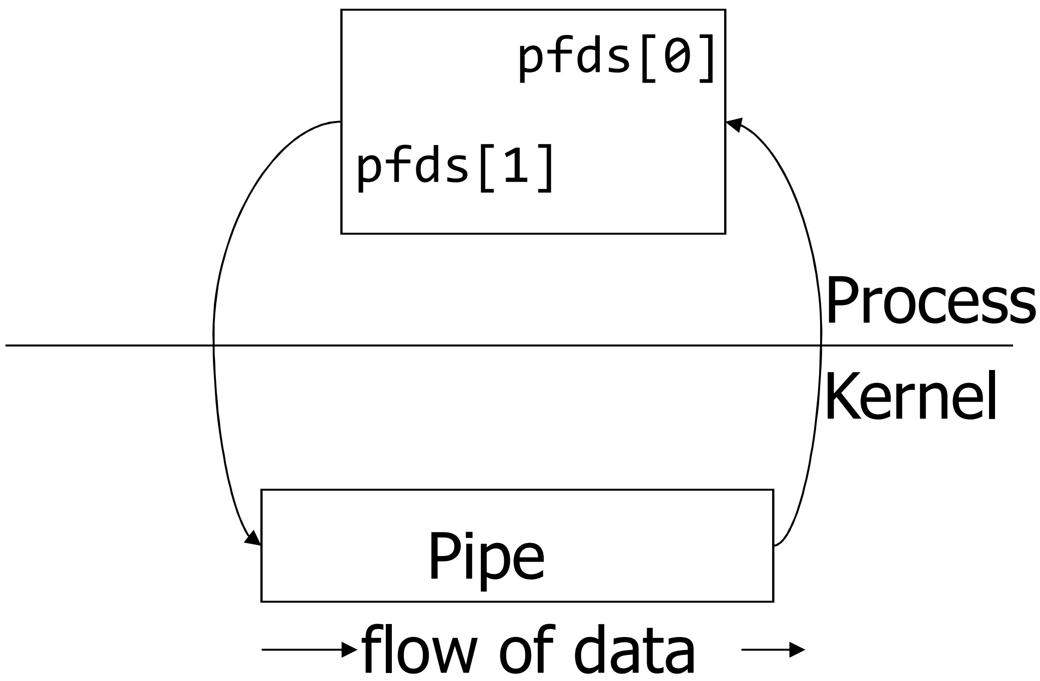
  - Creates a pair of file descriptors pointing to a pipe inode
  - Places them in the array pointed to by `filedes`
    - `filedes[0]` is for reading
    - `filedes[1]` is for writing
  - Return value: Success returns zero; error returns -1
- popen system call

```
FILE *popen(const char *command, const char *type);
FILE* file = popen("ntpdate", "r");
```

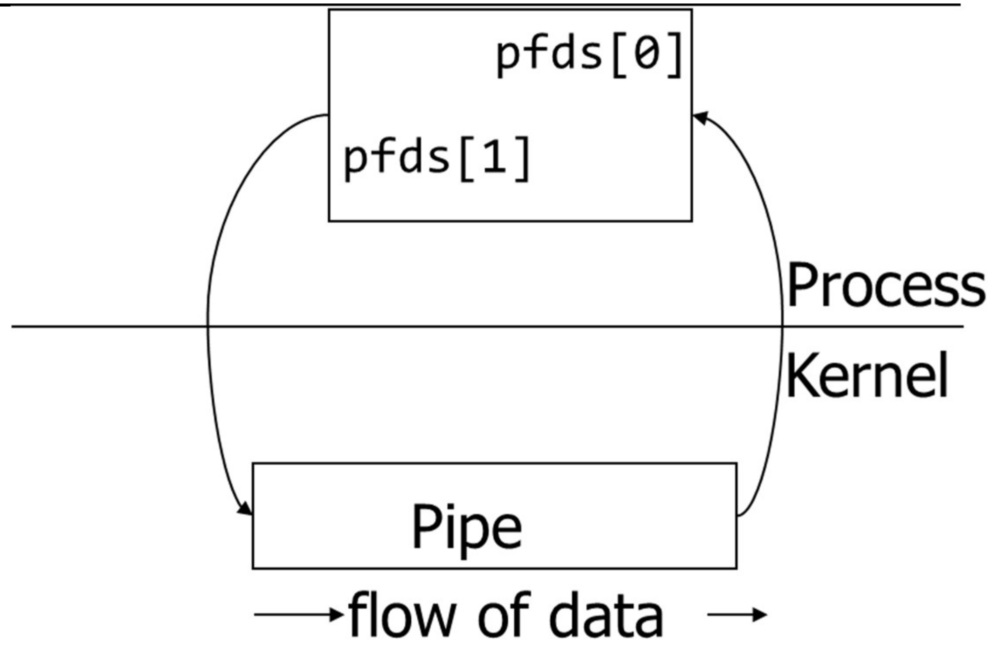
  - Opens a process by creating a pipe, forking, and invoking the shell

Pipe Creation

```
int main()
{
    int pfds[2];
    if (pipe(pfds) == -1)  {
        perror("pipe");
        exit(1);
    }
}
```



Pipe Example



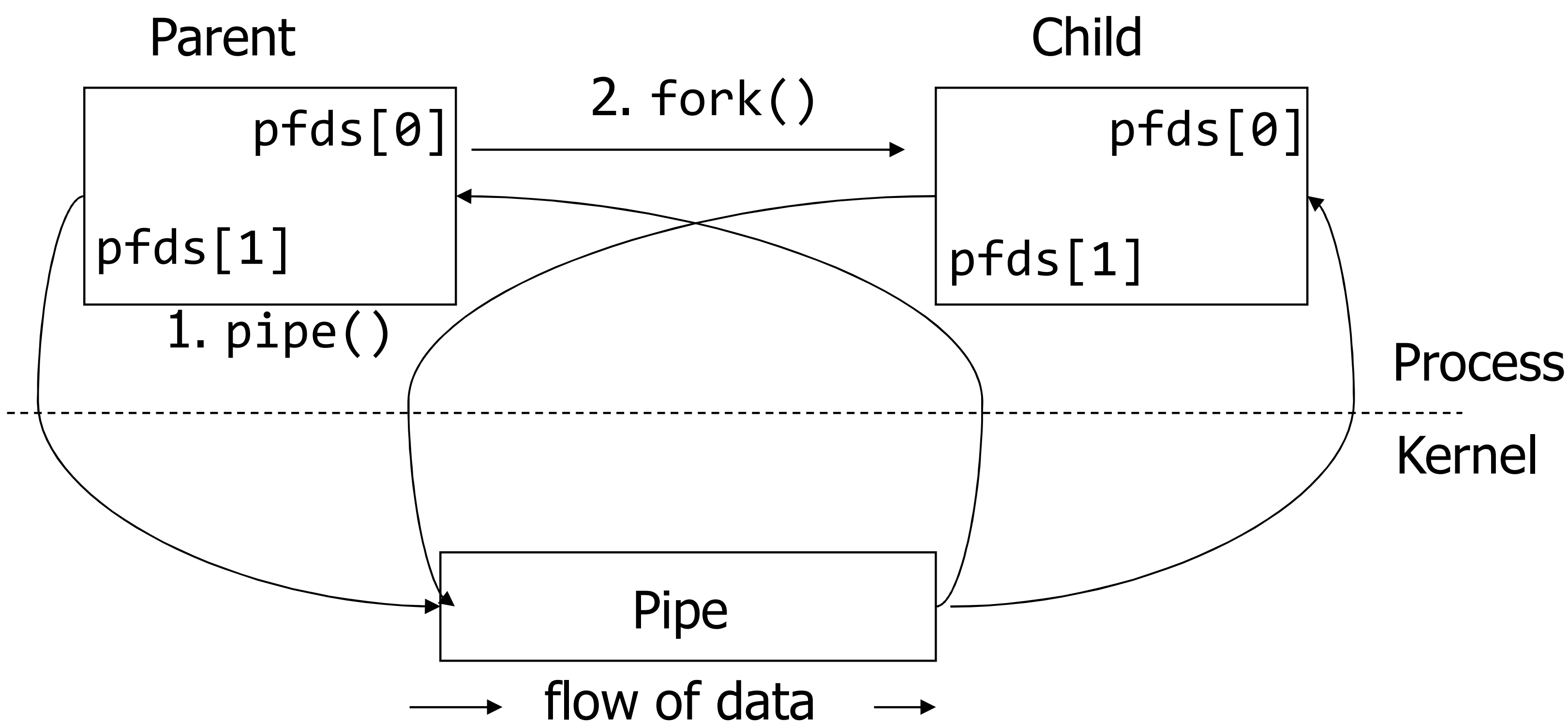

```
int main()
{
    int pfds[2];
    if (pipe(pfds) == -1)  {
        perror("pipe");
        exit(1);
    }
    printf("writing to file descriptor #%d\n", pfds[1]);
    write(pfds[1], "test", 5);
    printf("reading from file descriptor #%d\n",pfds[0]);
    read(pfds[0], buf, 5);
    printf("read %s\n", buf);
}
```

A Channel Between Parent and Child

- Child is created by a `fork()` call executed by the parent
  - Child process is an image of the parent process
  - All the file descriptors that are opened by the parent are available in the child

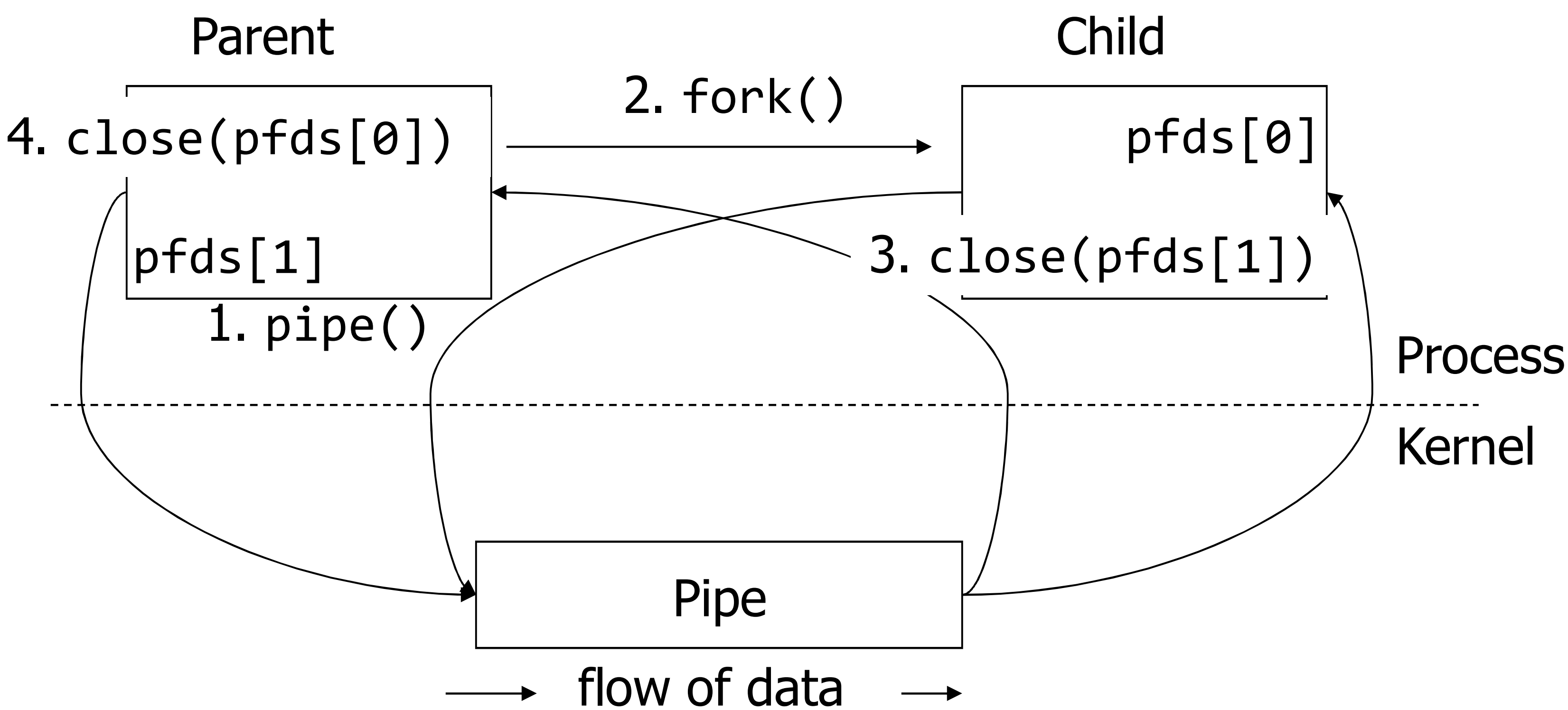
- Pipe is inherited by the child
  - File descriptors refer to the same I/O entity
  - Pipe may be passed on to the grand-children by the child process or other children by the parent

Piping Between Parent and Child



Piping Between Parent and Child

- To allow one way communication each process should close one end of the pipe



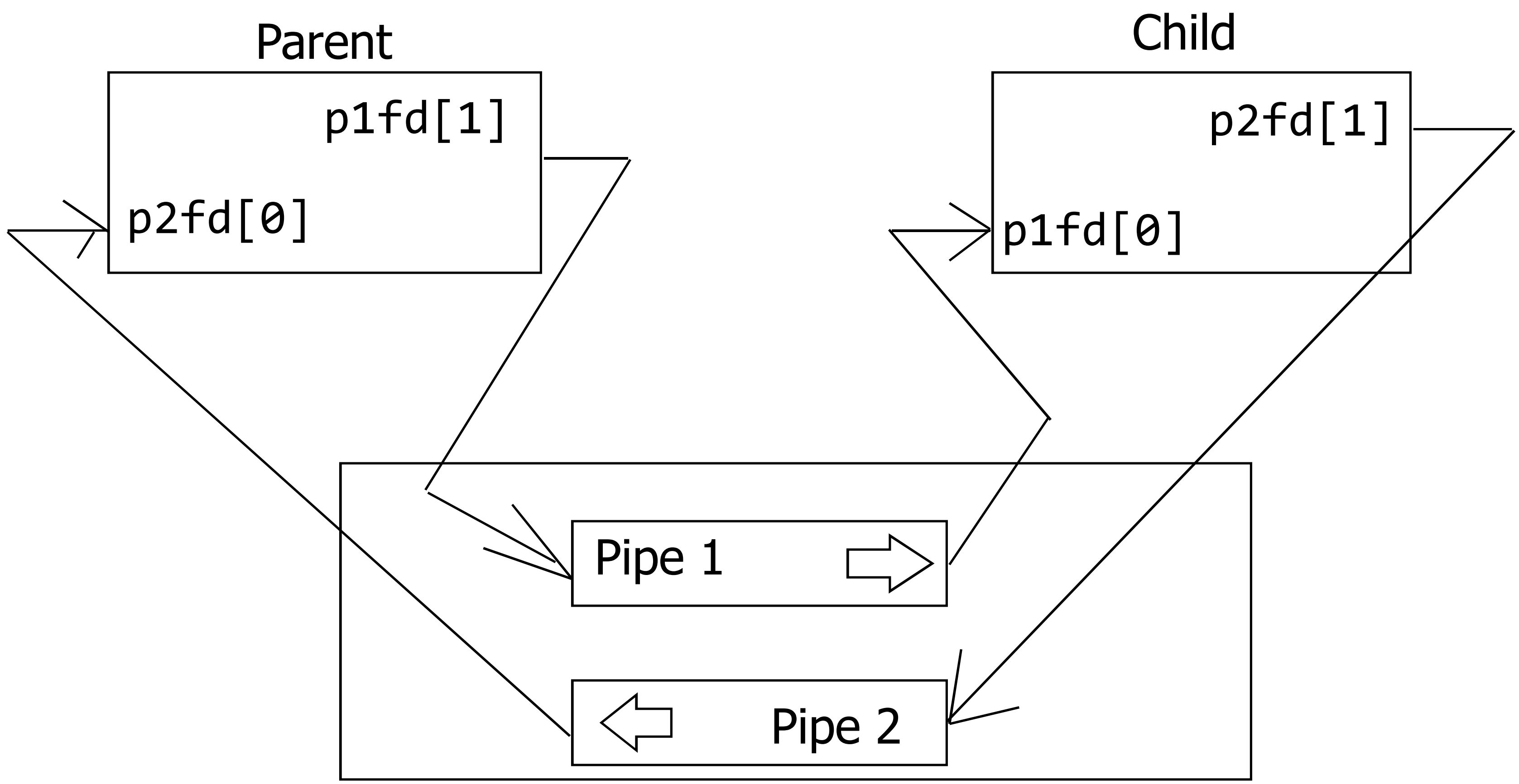
Pipe Closing

- The file descriptors associated with a pipe can be closed with the `close(fd)` system call
- A pipe exists until both file descriptors are closed in all processes
- How would we achieve two way communication?

Pipe Example

```
int main() {
    int pfds[2];  char buf[30];
    pipe(pfds);
    if (!fork()) {
        close(pfds[0]);
        printf(" CHILD: writing to the pipe\n");
        write(pfds[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    }
    else {
        close(pfds[1]);
        printf("PARENT: reading from pipe\n");
        read(pfds[0], buf, 5);
        printf( "PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }
}
```

Full Duplex Communication via Two Pipes



Named vs. Unnamed Pipes

Unnamed pipe

- Unnamed pipes can only be used between related process, such as parent/child, or child/child process
- Unnamed pipes can exist only as long as the processes using them

Named pipe

- When created, named pipes have a directory entry
  - Have file access permissions for unrelated processes to use the pipe
- Named pipes can be created by using `mkfifo` system call
  - `int mkfifo(const char *pathname, mode_t mode);`
  - Makes a FIFO special file with name `pathname` and FIFO's permissions
- Any process can open FIFO special file for reading or writing
  - Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa

Named Pipe Example

Process 1

```
int main() {
   int fd;
   char * myfifo = "/tmp/myfifo";
   /* create the FIFO(named pipe)*/
   mkfifo(myfifo, 0666);

   /* write "Hi" to the FIFO */
   fd = open(myfifo, O_WRONLY);
   write(fd, "Hi", sizeof("Hi"));
   close(fd);

   /* remove the FIFO */
   unlink(myfifo);

   return 0;
}
```

Process 2

```
#define MAX_BUF 1024

int main() {
   int fd;
   char * myfifo = "/tmp/myfifo";
   char buf[MAX_BUF];

   /* open, read, and display the
      message from the FIFO */
   fd = open(myfifo, O_RDONLY);
   read(fd, buf, MAX_BUF);
   printf("Received: %s\n", buf);
   close(fd);

   return 0;
}
```

Redirecting Standard I/O

- Redirection of `stdin`, `stdout` and `stderr` in Unix and Linux

<< or 0<<	redirect `stdin` within command
> or 1>	redirect `stdout` to file (overwrite if file not empty)
>> or 1>>	redirect `stdout` to file (append if file not empty)
2>	redirect `stderr` to file (overwrite if file not empty)
2>>	redirect `stderr` to file (append if file not empty)

- Redirecting data from one program to another
  - `ls | head -3`
  - `ls | head -3 | tail -1`
  - `ls | head -3 | tail -1 > myoutput`

Redirecting Standard I/O

Program

```
main( int ac, char *av[]){
   int i;
   printf ("No. of args: %d, Args:\n", ac);
   for (i=0; i < ac; i++)
      printf ("args[%d]: %s\n", i, av[i]);

   fprintf (stderr, "Msg sent to stderr.\n");
}
```

```
$ listargs > output 2> error

$ cat output
No. of args: 1, Args:
args[0]: listargs

$ cat error

Msg sent to stderr.

$ listargs arg1 > output 2>error

$ cat output
No. of args: 2, Args:
args[0]: listargs
args[1]: arg1
```

Redirection in C Programs

- Processes do not read from files, they read from file descriptors

- Close / Open Method:

```
close(0);                      /* file descriptor 0 – i.e. stdin */
open(filename, O_RDONLY);      /* redefines lowest file descriptor */
```

- Open / Close / Dup / Close / Method:

```
fd=open(filename, O_RDONLY);  /* open file to read - fd */
close(0);                     /* close file descriptor 0 – i.e.
                                 stdin */
newfd=dup(fd);                /* make "clone" of fd and
                                 uses lowest next available fd –
                                 I.e. 0 that was closed */
close(fd);                    /* close original fd associated
                                 with file. Now, stdin is
                                 associated with file…   */
```

dup2 is similar to dup, but dup2 will automatically `close(0)`.
For example, `newfd=dup2(fd,0);`

Any Question So Far?