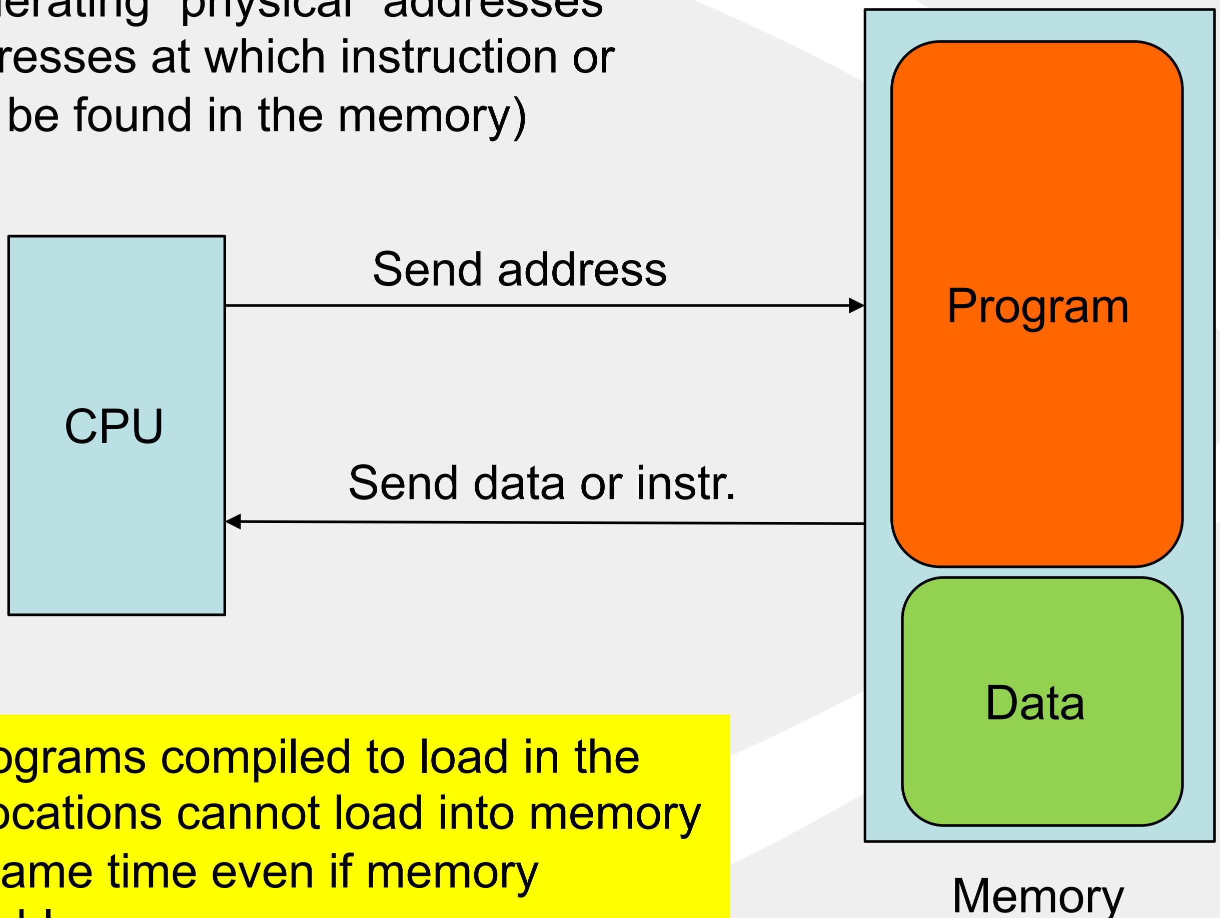
Simple Computer

CPU generating “physical” addresses (i.e., addresses at which instruction or data can be found in the memory)



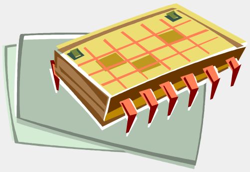
Two programs compiled to load in the same locations cannot load into memory at the same time even if memory is available.

Simple Computer

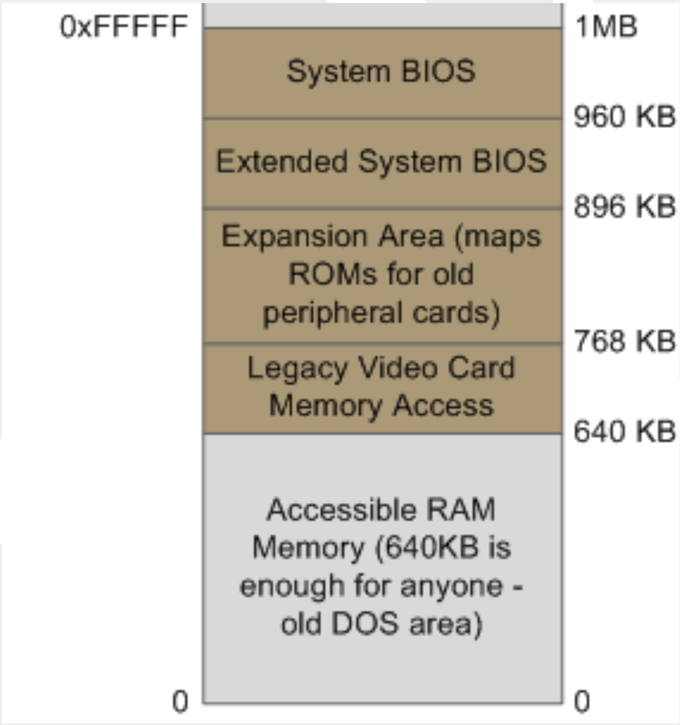
- Consider a computer running in "real" mode – no virtual addresses (i.e., physical addresses are used)

<u>Power-On-Reset:</u>

- CPU goes to a fixed location on power on
- BIOS (ROM)
- Perform diagnostics and loads the actual OS loader
- OS loader resident in secondary storage
- OS loader loads the actual OS or loads other programs



CPU



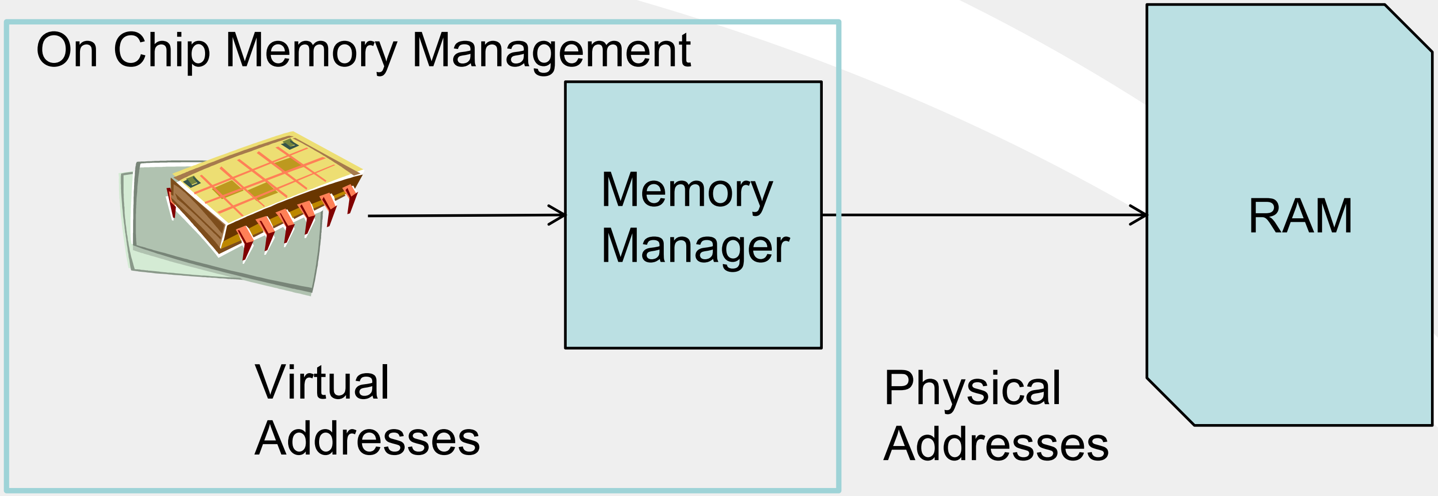
Simple Computer…

- Nothing is virtualized (very similar to DOS)
- A resident program prevents loading a newer program even when space is available
- Easy to debug – memory inspection possible

Simple Computer...

- Modern Microprocessors (upwards of i386) provide virtual addressing
- What is virtual addressing?
  - mov $0x8048570,%eax
  - (move register eax to memory location)
  - (the location actually accessed in the physical RAM is different – it is defined by mapping tables)

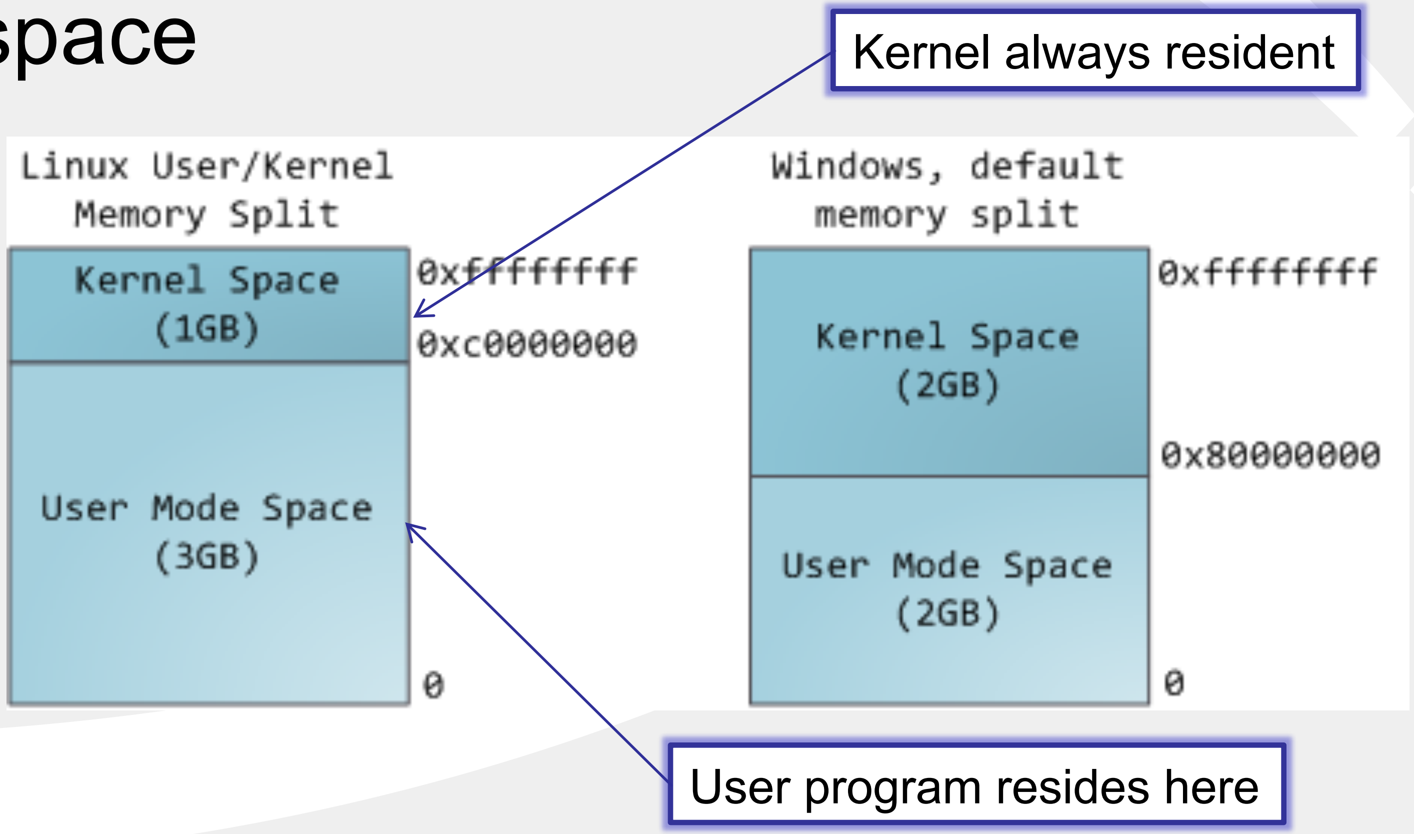
Idea 1: Address Virtualization



- Leverage Virtual Addresses
  - All addresses dealt with the processor are virtual
  - Translated by a table-based translation mechanism to physical addresses

Address Spaces

- Instead of sharing the memory space – give each process the full address space

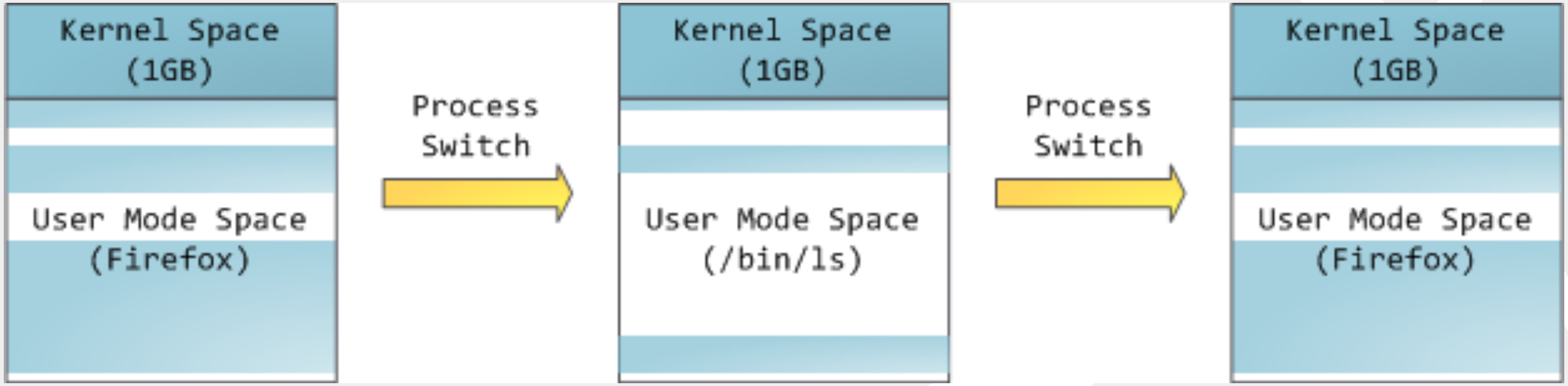


Address Space

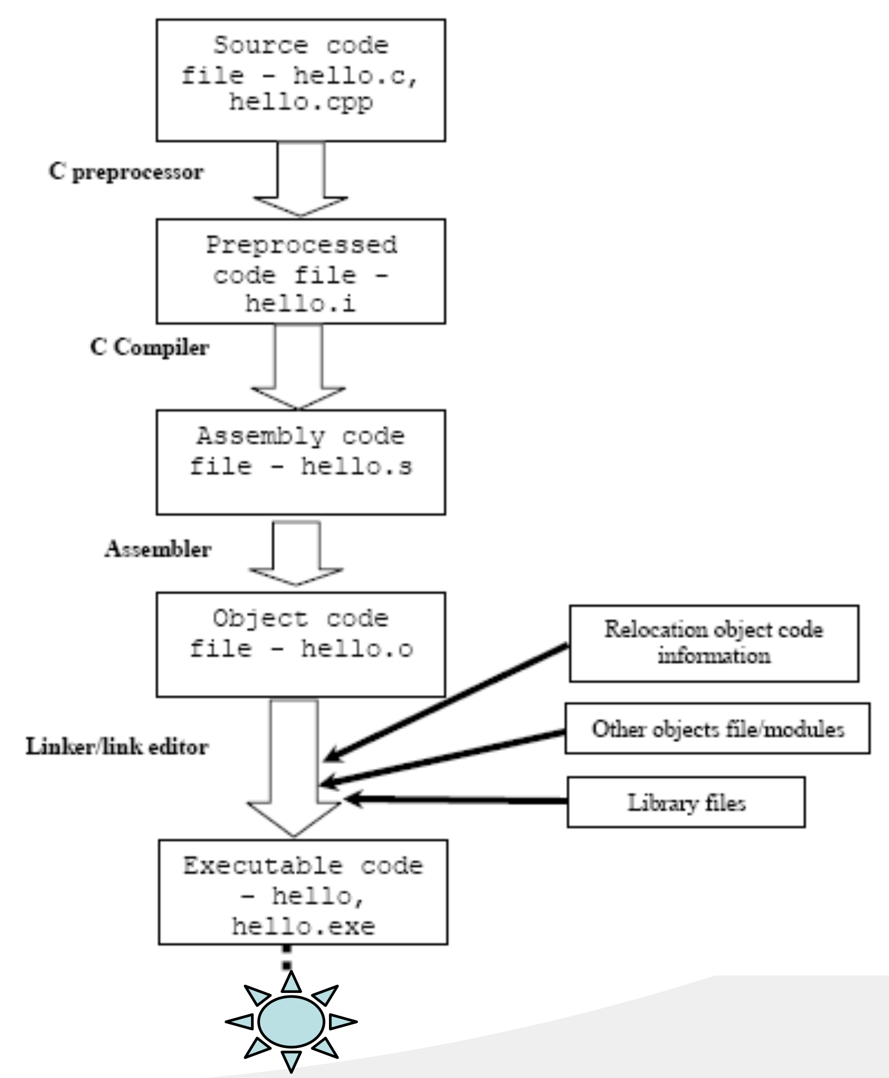
- When a process runs, all the addresses it could generate (full address space) belong to it – not shared with any other process
- Catch – part of it is taken by the kernel – the space mapped to the kernel – is persistent
- Using the kernel space – a process could communicate with other processes, how?

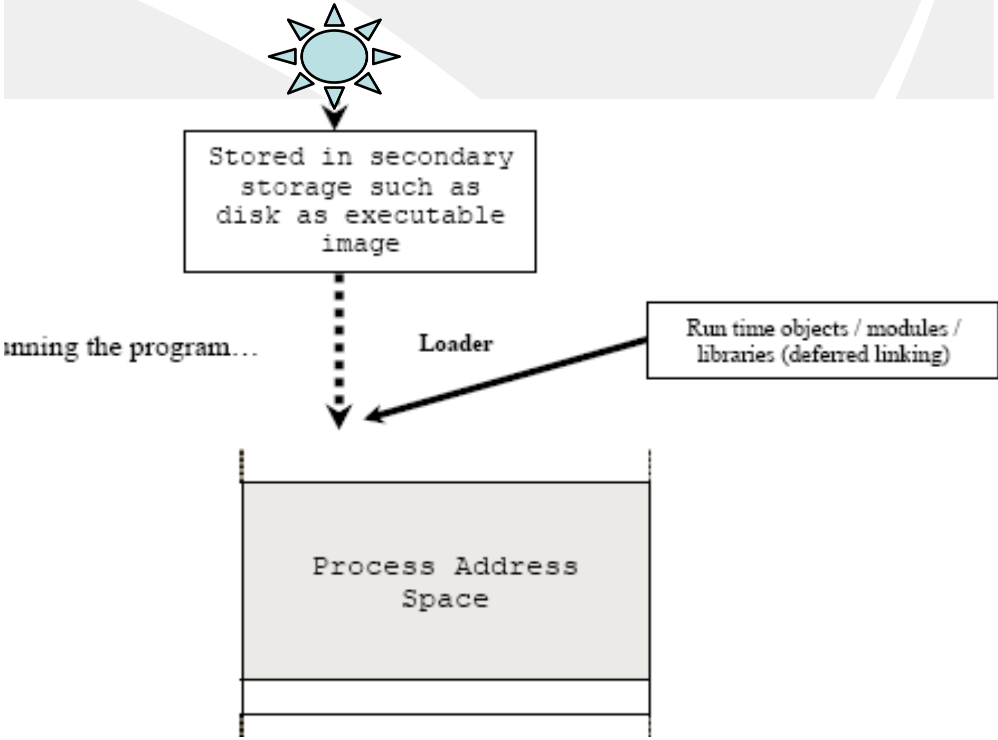
Address Space Switching

- Address space switching happens with a process switch

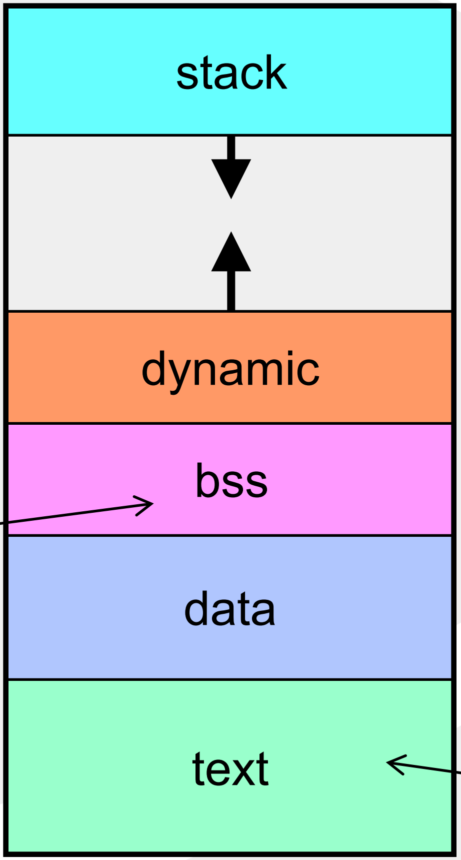


Program to Process





Address Space: Details of the User Space

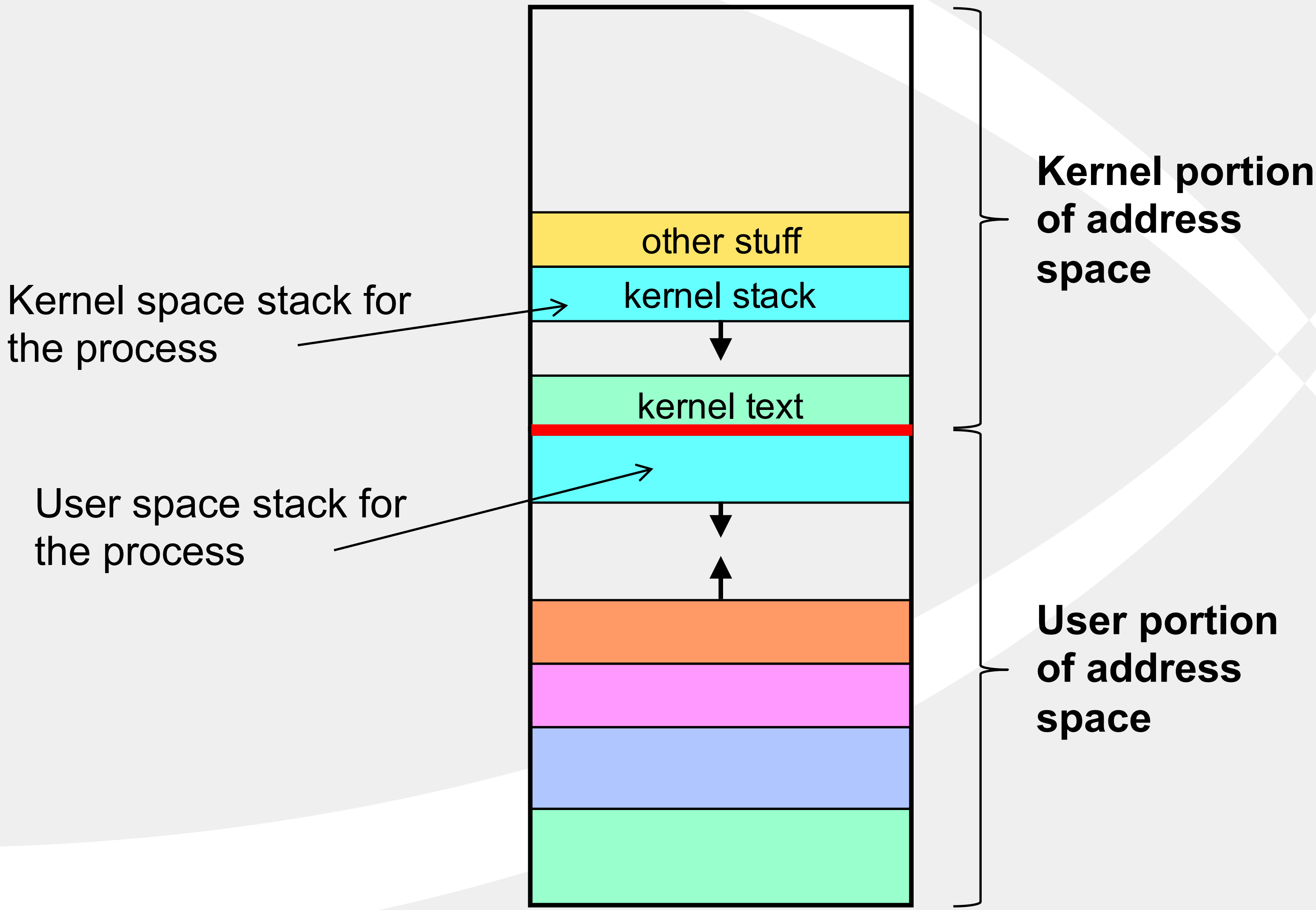


Block Started by Symbol: Includes space for uninitialized variables

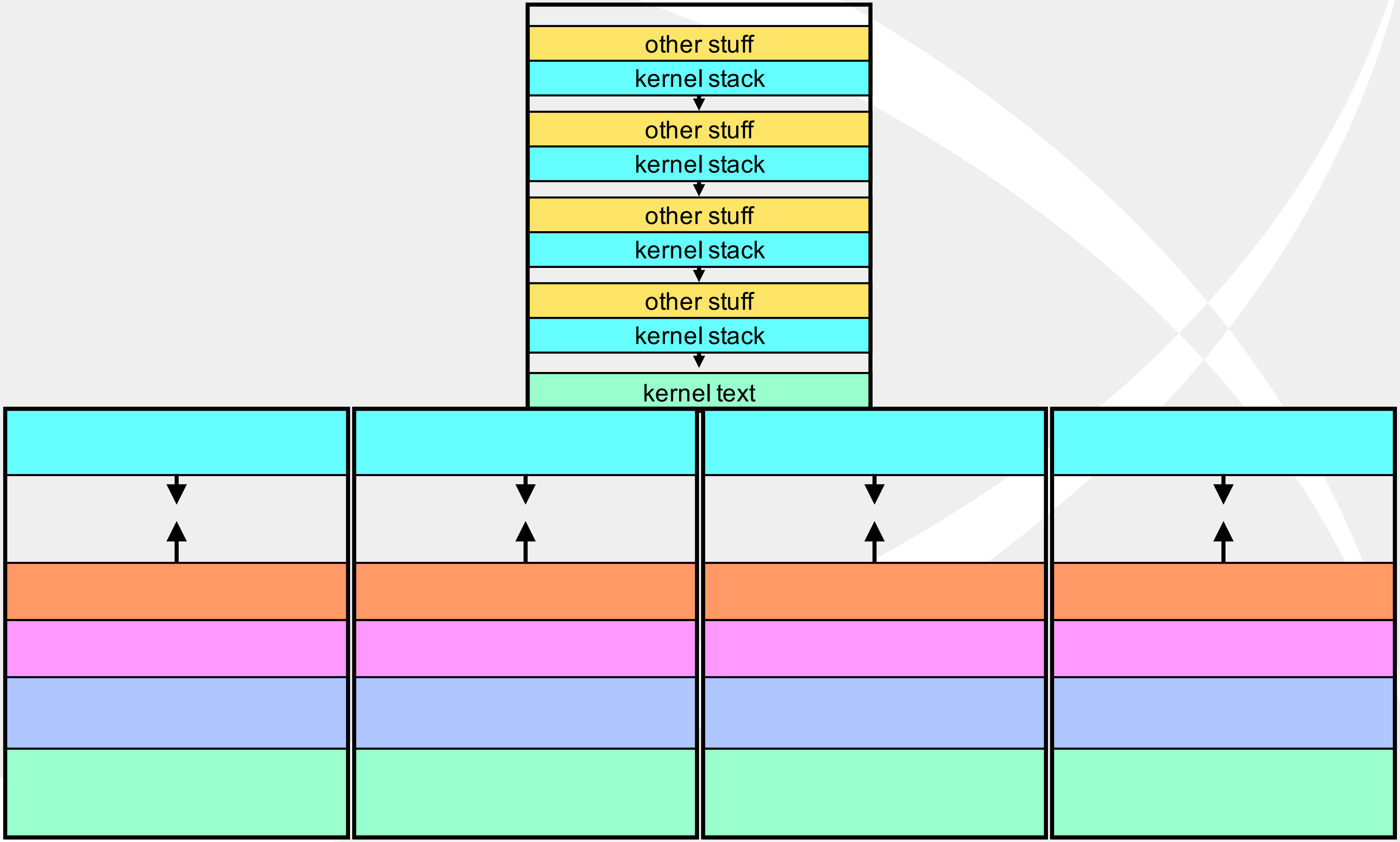
Read only segment that contains the program instructions

Address space occupied by user programs when **address space is NOT randomized**

Process Address Space



Multiple Processes



What is a *process*?

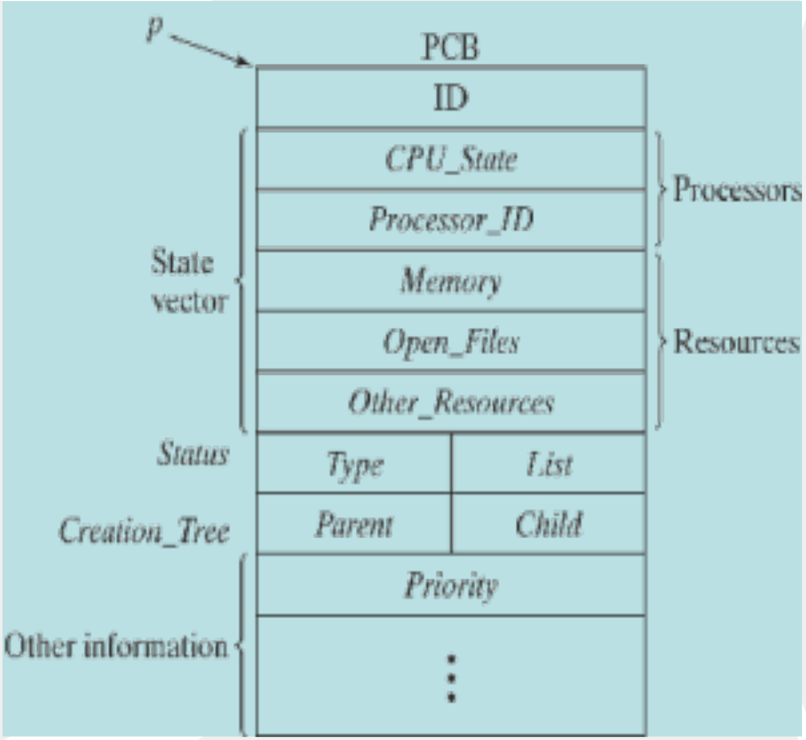
- Is a process the same as a program? No!, it is both; *more* and *less*
  - *more*—a program is just part of a process context.
    - `tar` can be executed by two different people—same program (shared code) as part of different processes.
  - *less*—a program may invoke several processes.
    - `cc` invokes `cpp`, `cc1`, `cc2`, `as`, and `ld`.

Process management issues?

- Process management issues:
  - Lifecycle management
  - Precedence management (flow management)
- Lifecycle management:
  - Process creation
  - Process state changes, reasons (what happens in the middle!)
  - Process termination

How is a process represented?

- Information: state & control
- Process Control Block (PCB)
  - Identifier
  - State Vector = Information necessary to run process p
  - Status
  - Creation tree
  - Priority
  - Other information



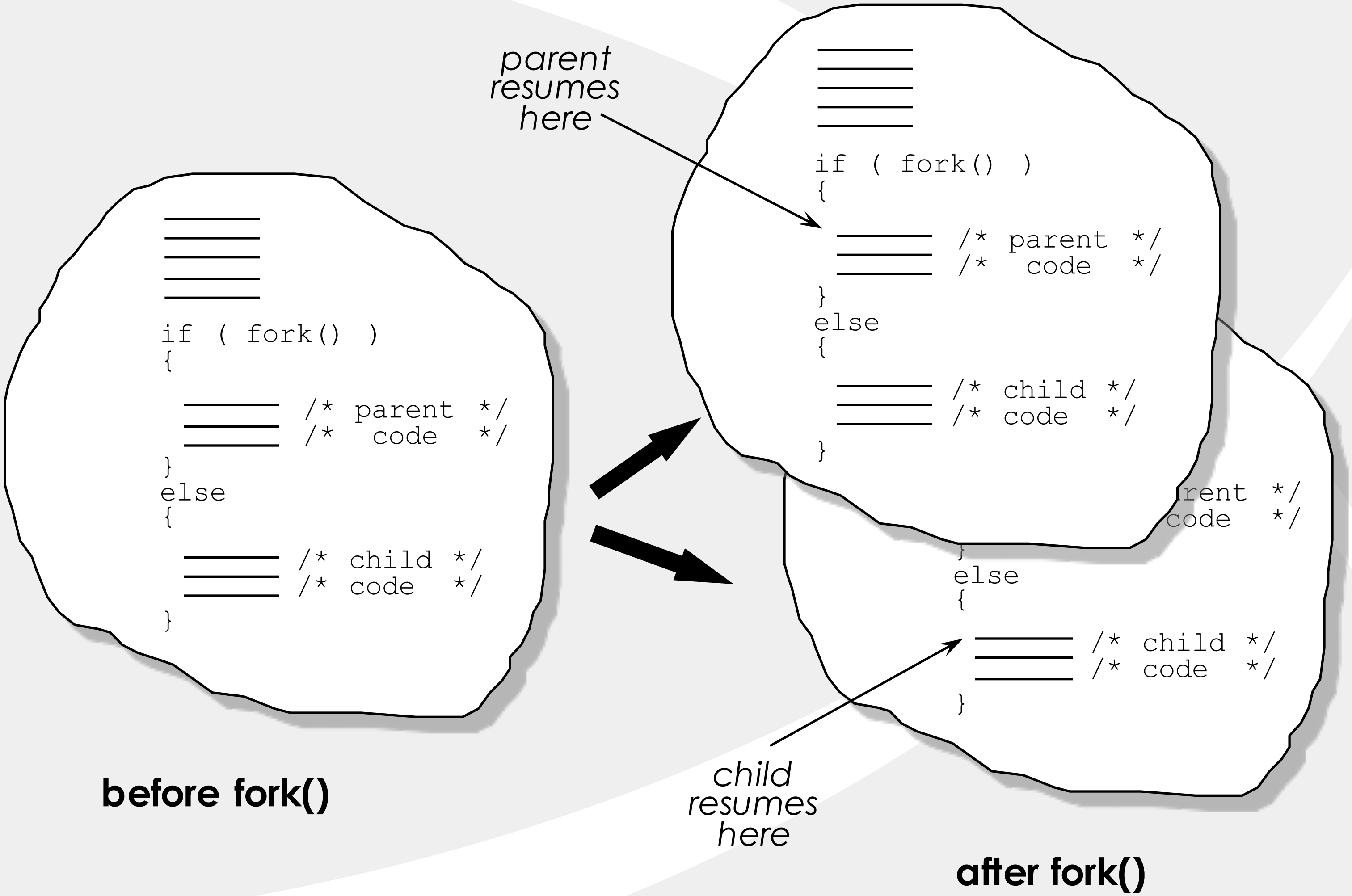
Lifecycle: Create process

- Two ways of creating a new process:
  - *Build one from scratch*:
    - load *code* and *data* into memory
    - create (empty) a *dynamic memory workspace (heap)*
    - create and initialize the *process control block*
    - make process known to process scheduler (dispatcher)
  - *Clone an existing one*:
    - stop current process and save its state
    - make a copy of *code*, *data*, *heap* and *process control block*
    - make process known to process scheduler (dispatcher)

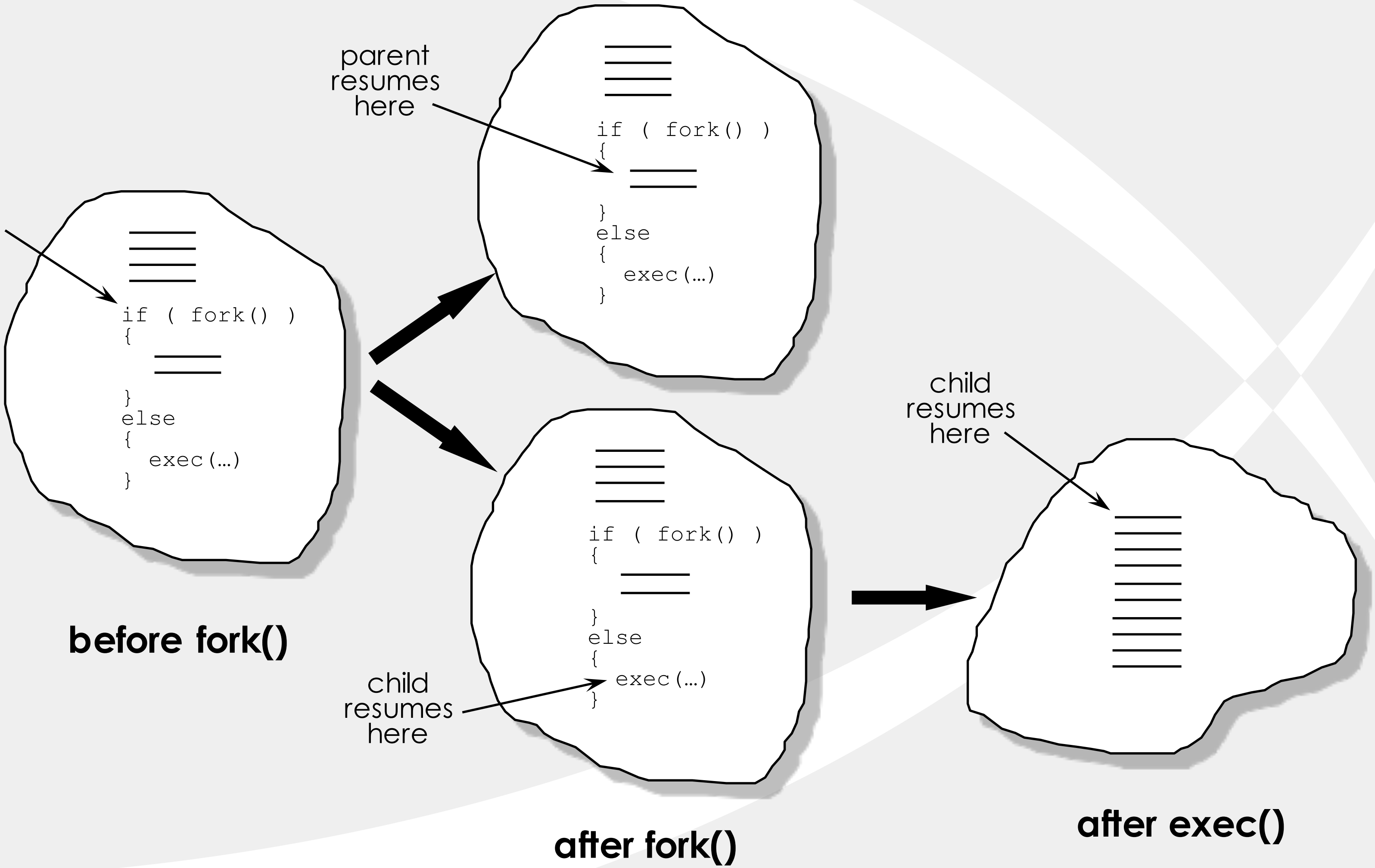
UNIX process creation

- In UNIX, the **`fork()`** system call is used to create processes
  - **`fork()`** creates an identical copy of the calling process
  - after the **`fork()`**, the *parent* continues running concurrently with its *child* competing equally for the CPU

UNIX process creation...



A typical use of `fork()`

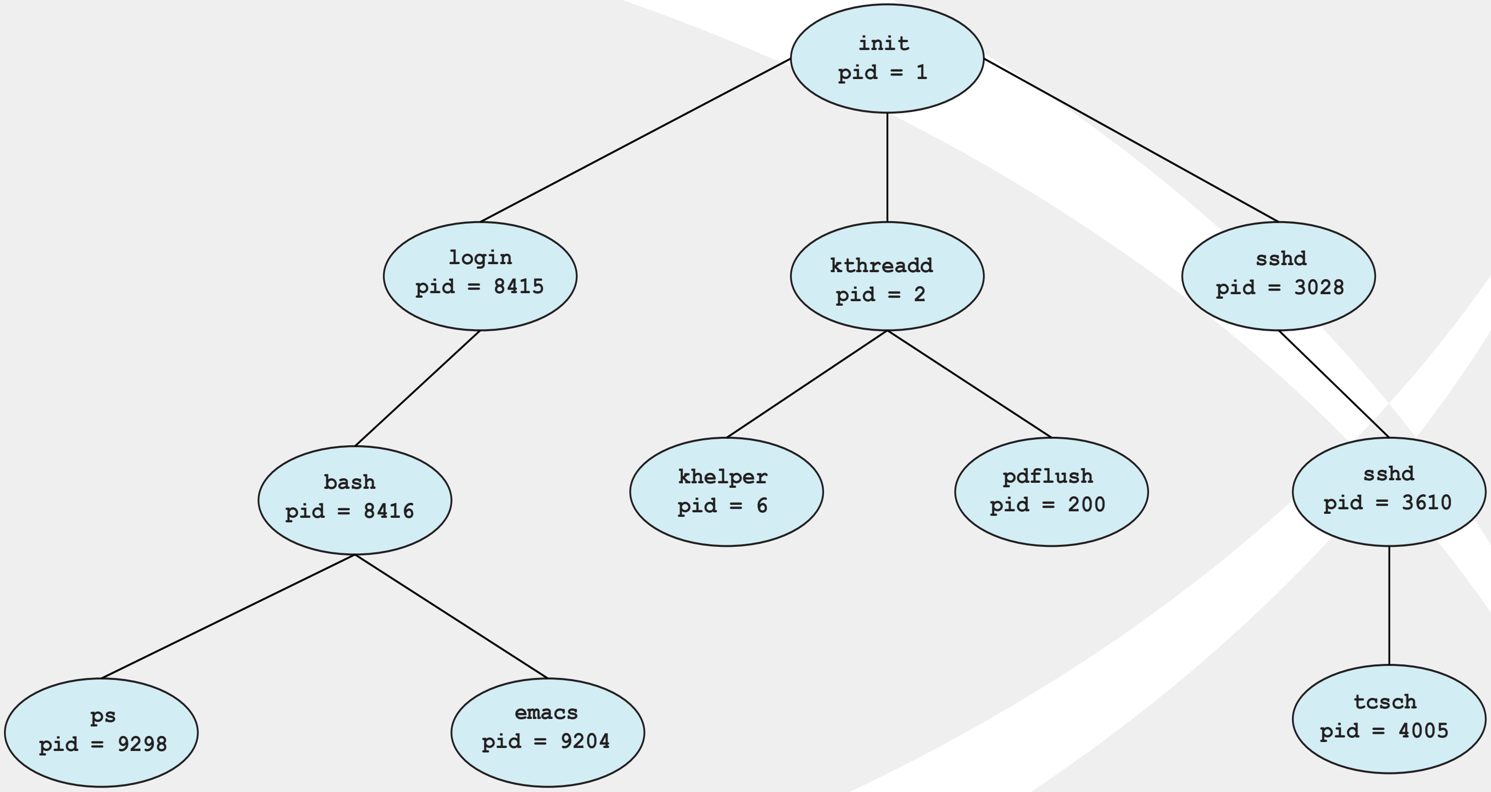


Example

- What is the output of the following simple C program?

```
main() {
  int i;
  i = 10;
  if (fork() == 0) i += 20;
  printf(" %d ", i);
}
```

A Tree of Processes in Linux



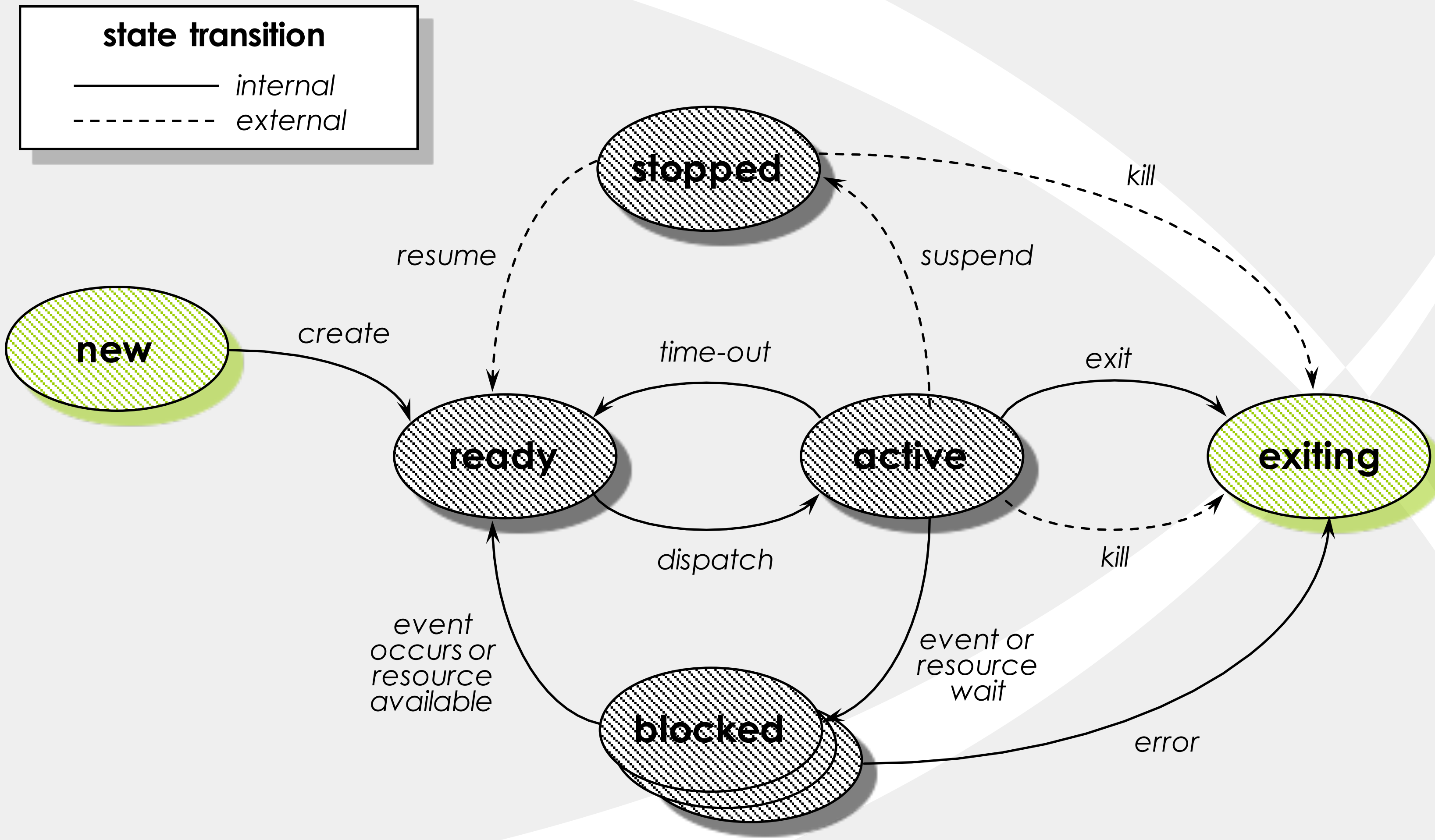
Lifecycle: After creation

- After creation, process can experience various conditions:
  - No resources to run (e.g., no processor, memory)
  - Waiting for a resource or event
  - Completed the task and exit
  - Temporarily suspend waiting for a condition
- → Process should be in different states

Process states

- A process can be in many different states:
  - **New**—a process being created but not yet included in the pool of executable processes (*resource acquisition*)
  - **Ready**—processes are prepared to execute when given the opportunity
  - **Active**—the process that is currently being executed by the CPU
  - **Blocked**—a process that cannot execute until some event occurs
  - **Stopped**—a special case of **blocked** where the process is suspended by the operator or the user
  - **Exiting**—a process that is about to be removed from the pool of executable processes (*resource release*)

Process state diagram



Example

- Following are code segments from a process that is already created and eligible to run or running

```
     ….
(a)  i = i + j * 10;
     a[i] = b[j] * c[i];
(b)  read(scale);          // reading standard
     input
                           for a variable
     ….
(c)  wait (mutex);     // waiting on a mutual
                       exclusion variable
```

- What are the possible process states at (a), (b), and (c)?

Lifecycle: Process termination

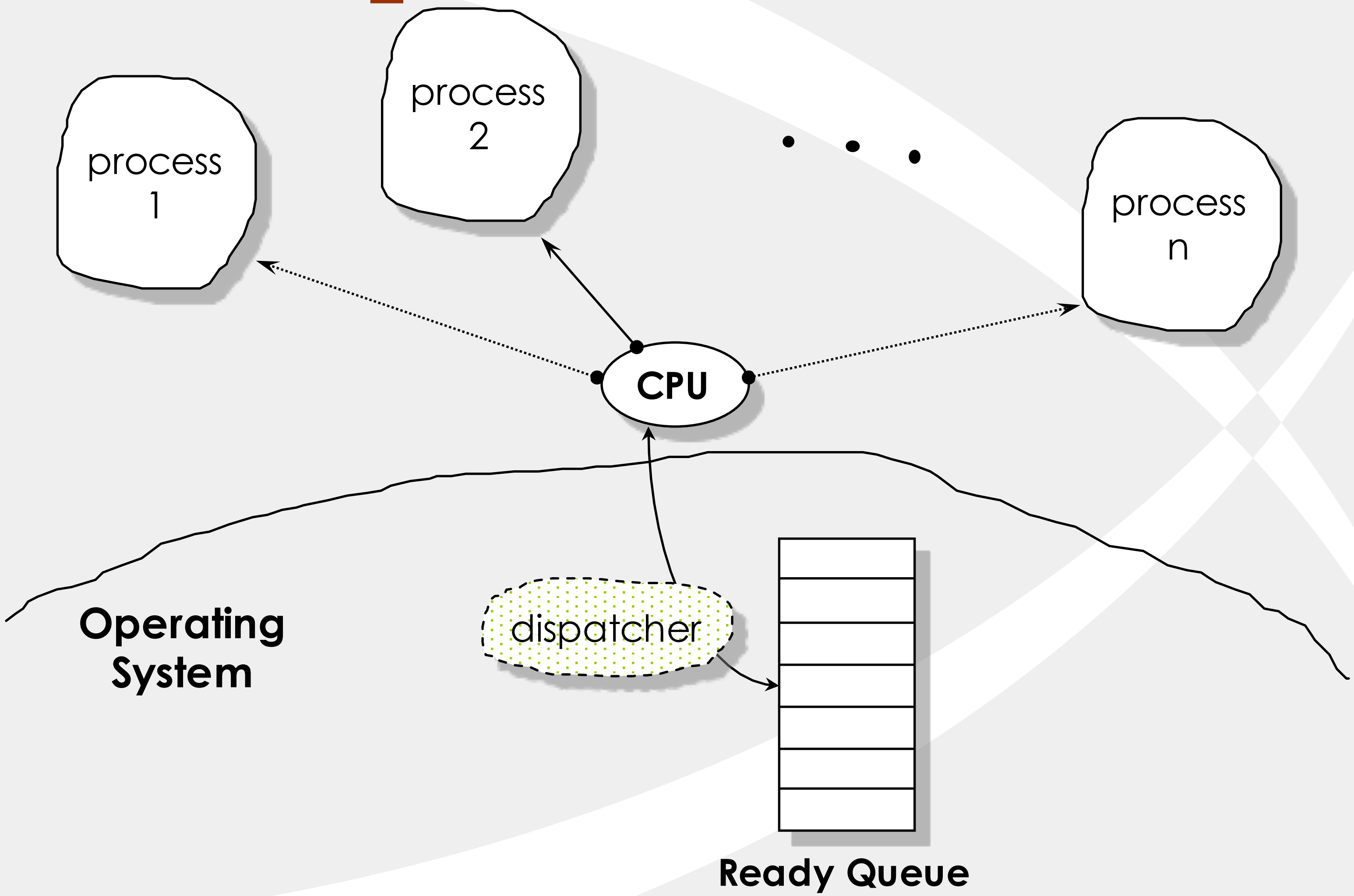
- A process enters the *exiting* state for one of the following reasons
  - normal completion: A process executes a system call for termination (e.g., in UNIX **`exit()`** is called).
  - abnormal termination:
    - programming errors
      - *run time*
      - *I/O*
    - user intervention

Implementing processes

- With multi-programming, we have several processes concurrently executing
- OS is responsible:
  - Dynamically selecting the next process to run
  - Rescheduling performed by the dispatcher
- Dispatcher given by:

```
loop forever {
        run the process for a while.
        stop process and save its state.
        load state of another process.
}
```

Dispatcher at work



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 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 dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

Dispatcher: Controlling the CPU?

- CPU can only do one thing at a time
- While user process running, dispatcher (OS) is NOT running
- How does the dispatcher regain control?
  - Trust the process to wake up the dispatcher when done (*sleeping beauty approach*).
  - Provide a mechanism to wake up the dispatcher (*alarm clock*).
- Obviously, the *alarm clock* approach is better. Why?

How is an alarm event handled?

- Context switch happens:
  - OS saves the state of the *active* process and restores the state of the *interrupt service routine*
  - Simultaneously, CPU switches to *supervisory* mode
- What must get saved? *Everything that the next process could or will damage*. For example:
  - *Program counter (PC)*
  - *Program status word (PSW)*
  - *CPU registers (general purpose, floating-point)*
  - *File access pointer(s)*
  - *Memory (perhaps?)*
- While saving the state, the operating system should mask (disable) *all* interrupts.

Memory: *to save* or *NOT to save*

- Here are the possibilities:
  - Save *all* memory onto disk.

    Could be *very* time-consuming. E.g., assume data transfers to disk at 1MB/sec. How long does saving a 4MB process take?

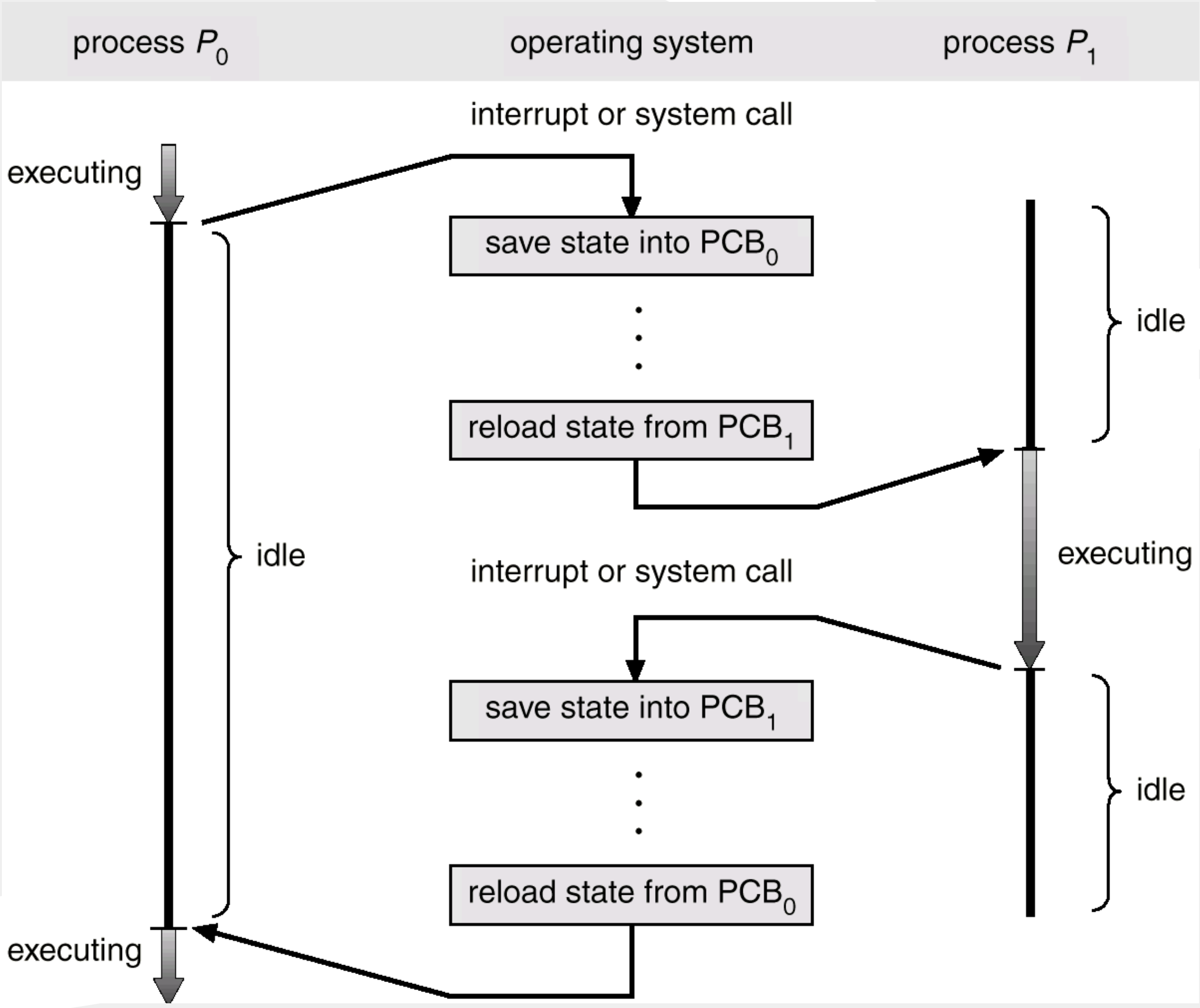
  - Don't save memory; trust next process.

    This is the approach taken by (older) PC and Mac OSes.

  - Isolate (protect) memory from next process.

    This is *memory management*, to be covered later

CPU switching among processes



Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- 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

Standard File Descriptors

```
main( ) {
   char buf[BUFSIZE];
   int n;
   const char* note = "Write failed\n";

   while ((n = read(0, buf, sizeof(buf))) > 0)
    if (write(1, buf, n) != n) {
             (void)write(2, note, strlen(note));
             exit(EXIT_FAILURE);
    }
   return(EXIT_SUCCESS);
}
```

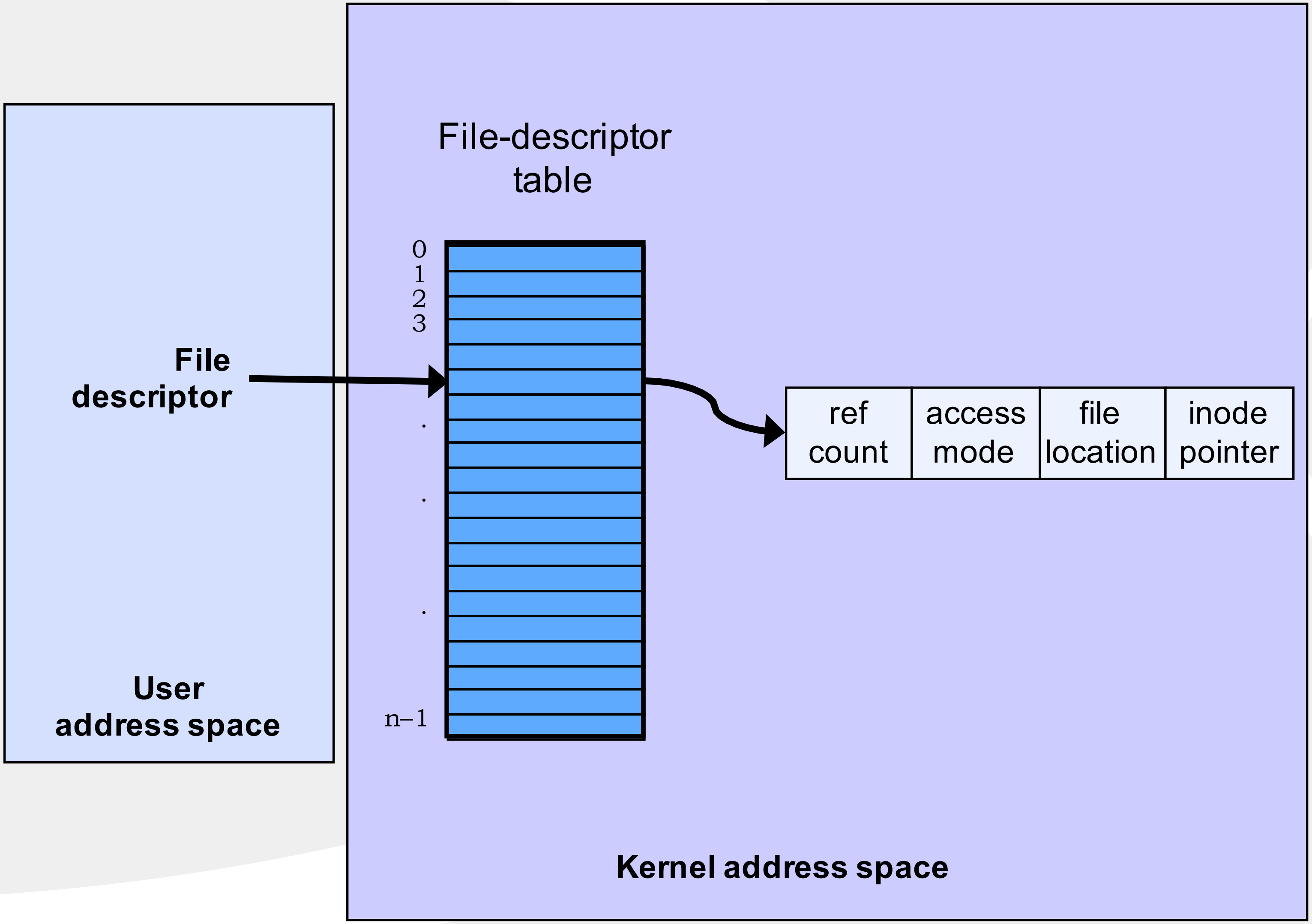
Running An Example File I/O

```
if (fork() == 0) {
    /* set up file descriptor 1 in the child process */
    close(1);
    if (open("/home/twd/Output", O_WRONLY) == -1) {
       perror("/home/twd/Output");
       exit(1);
    }
    execl("/home/twd/bin/primes", "primes", "300", 0);
    exit(1);
}

/* parent continues here */

while(pid != wait(0))      /* ignore the return code */
    ;
```

File-Descriptor Table



Allocation of File Descriptors

- Whenever a process requests a new file descriptor, the lowest numbered file descriptor not already associated with an open file is selected; thus

```
#include <fcntl.h>
#include <unistd.h>

close(0);
fd = open("file", O_RDONLY);
```

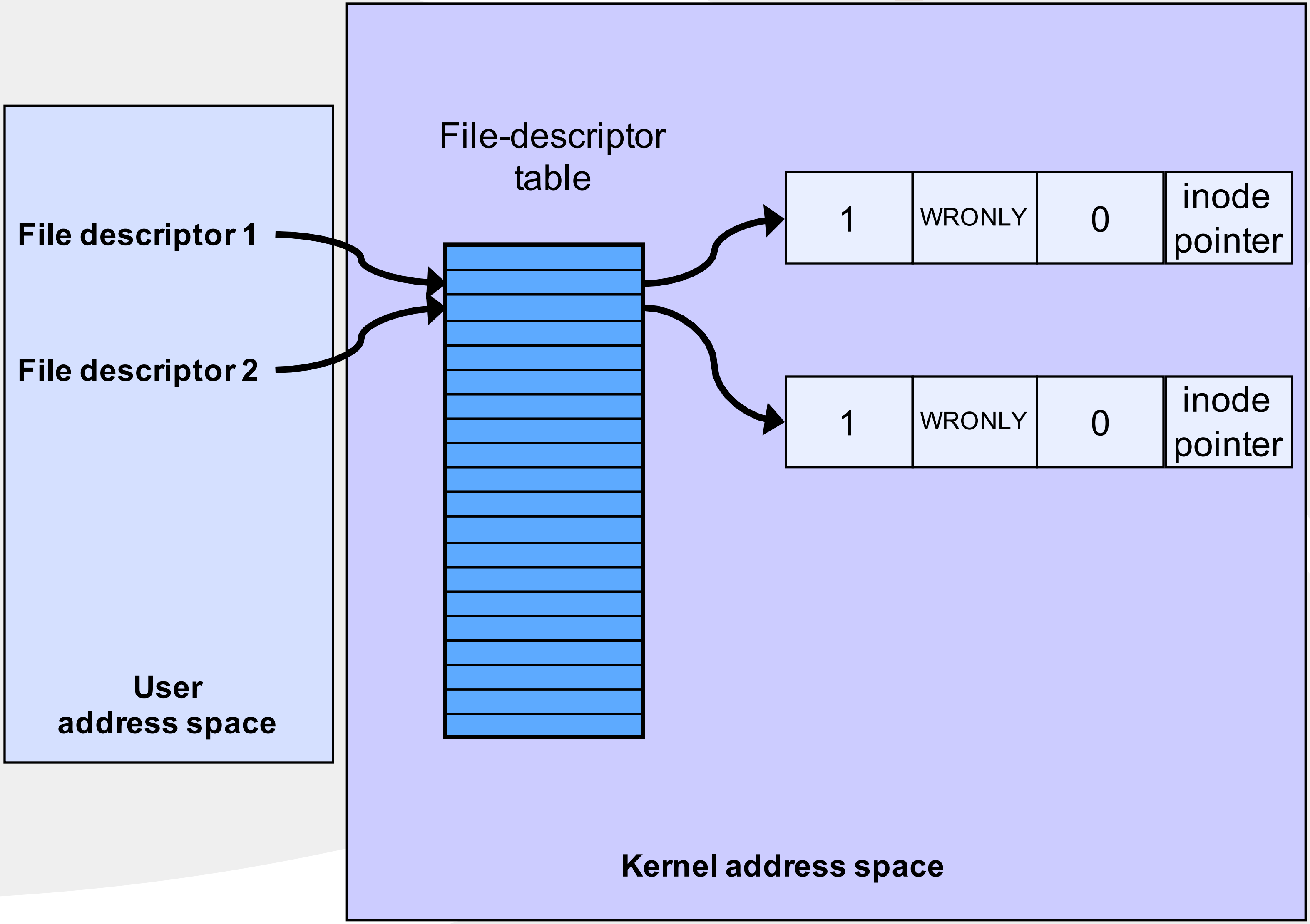
  - will always associate *file* with file descriptor 0 (assuming that the *open* succeeds)

Redirecting Output … Twice

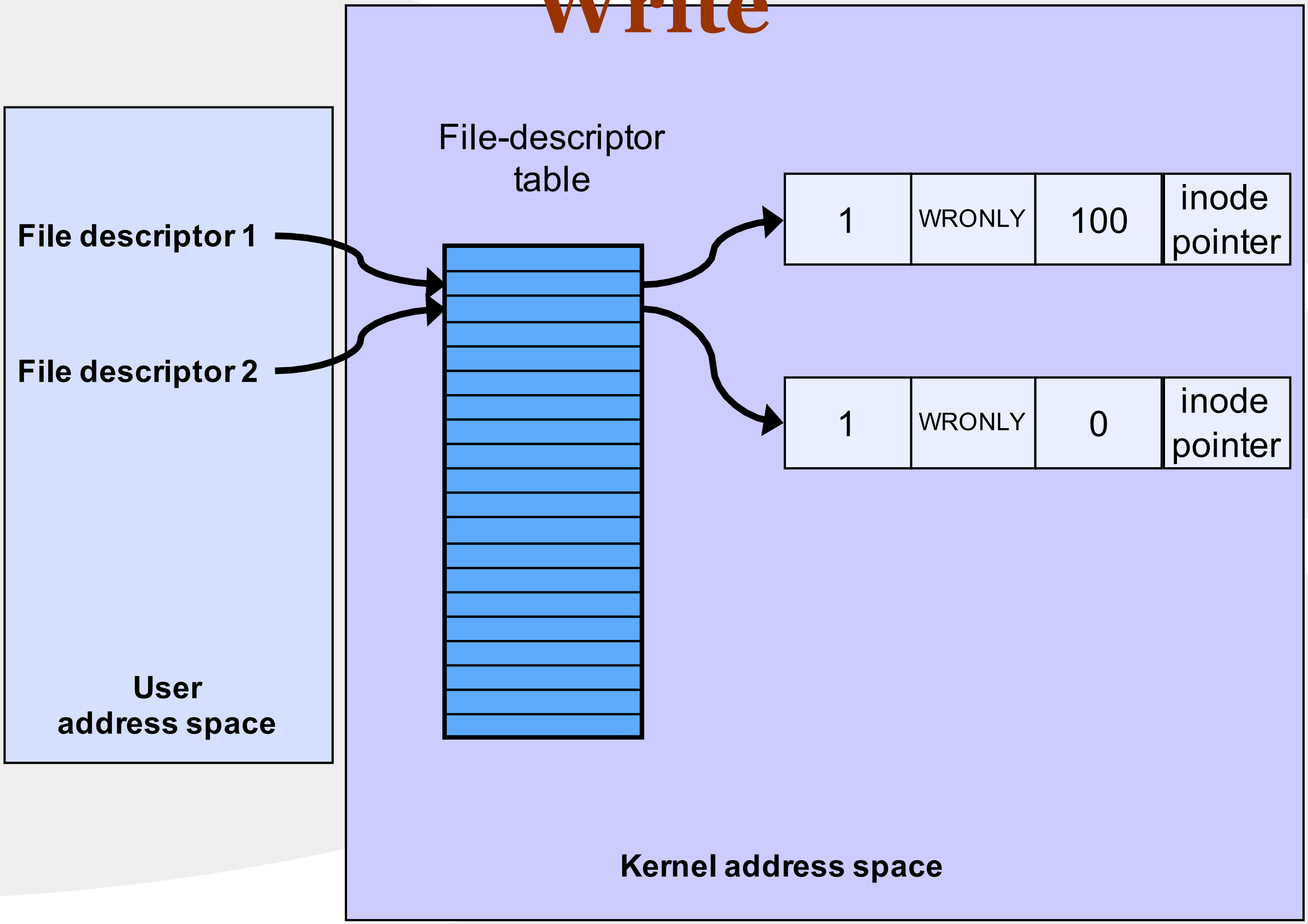
```
if (fork() == 0) {
   /* set up file descriptors 1 and 2 in the child process */
   close(1);
   close(2);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   execl("/home/twd/bin/program", "program", 0);
   exit(1);
}

/* parent continues here */
```

Redirected Output



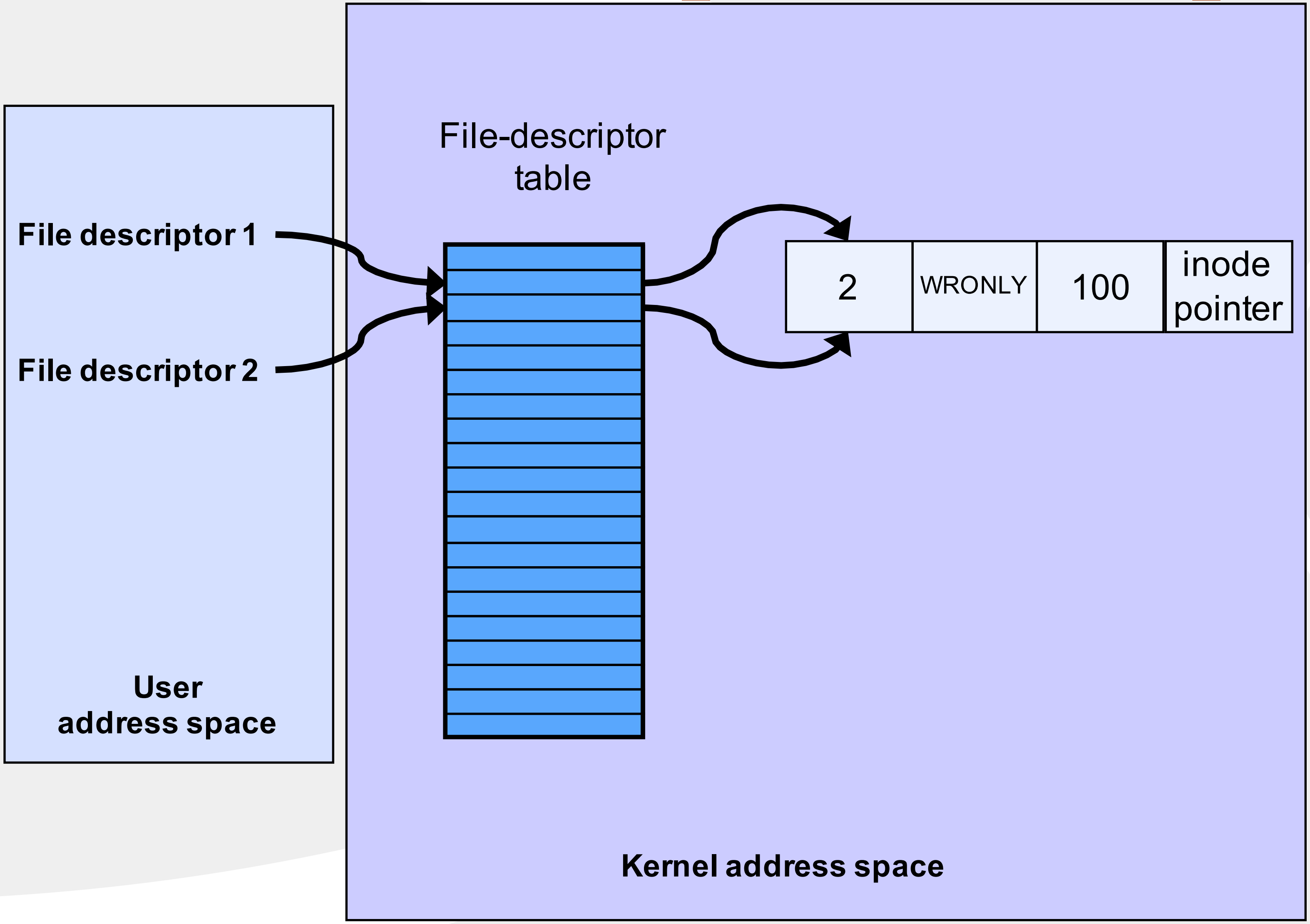
Redirected Output After Write



Sharing Context Information

```
if (fork() == 0) {
   /* set up file descriptors 1 and 2 in the child process */
   close(1);
   close(2);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   dup(1); /* set up file descriptor 2 as a duplicate of 1 */
   execl("/home/twd/bin/program", "program", 0);
   exit(1);
}
/* parent continues here */
```

Redirected Output After Dup



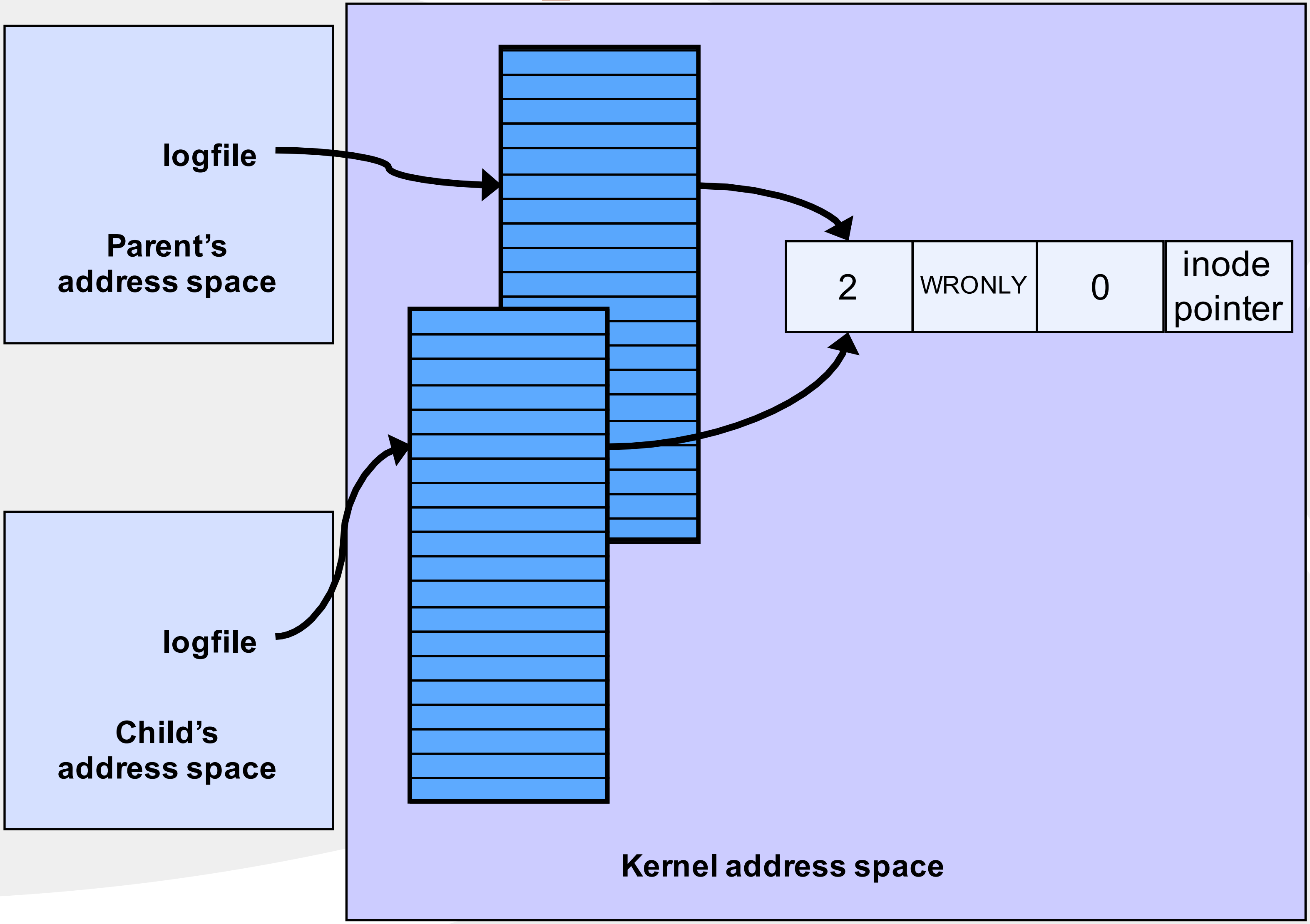
Fork and File Descriptors

```
int logfile = open("log", O_WRONLY);
if (fork() == 0) {
   /* child process computes something, then does: */
   write(logfile, LogEntry, strlen(LogEntry));
   …
   exit(0);
}

/* parent process computes something, then does: */

write(logfile, LogEntry, strlen(LogEntry));
…
```

File Descriptors After Fork

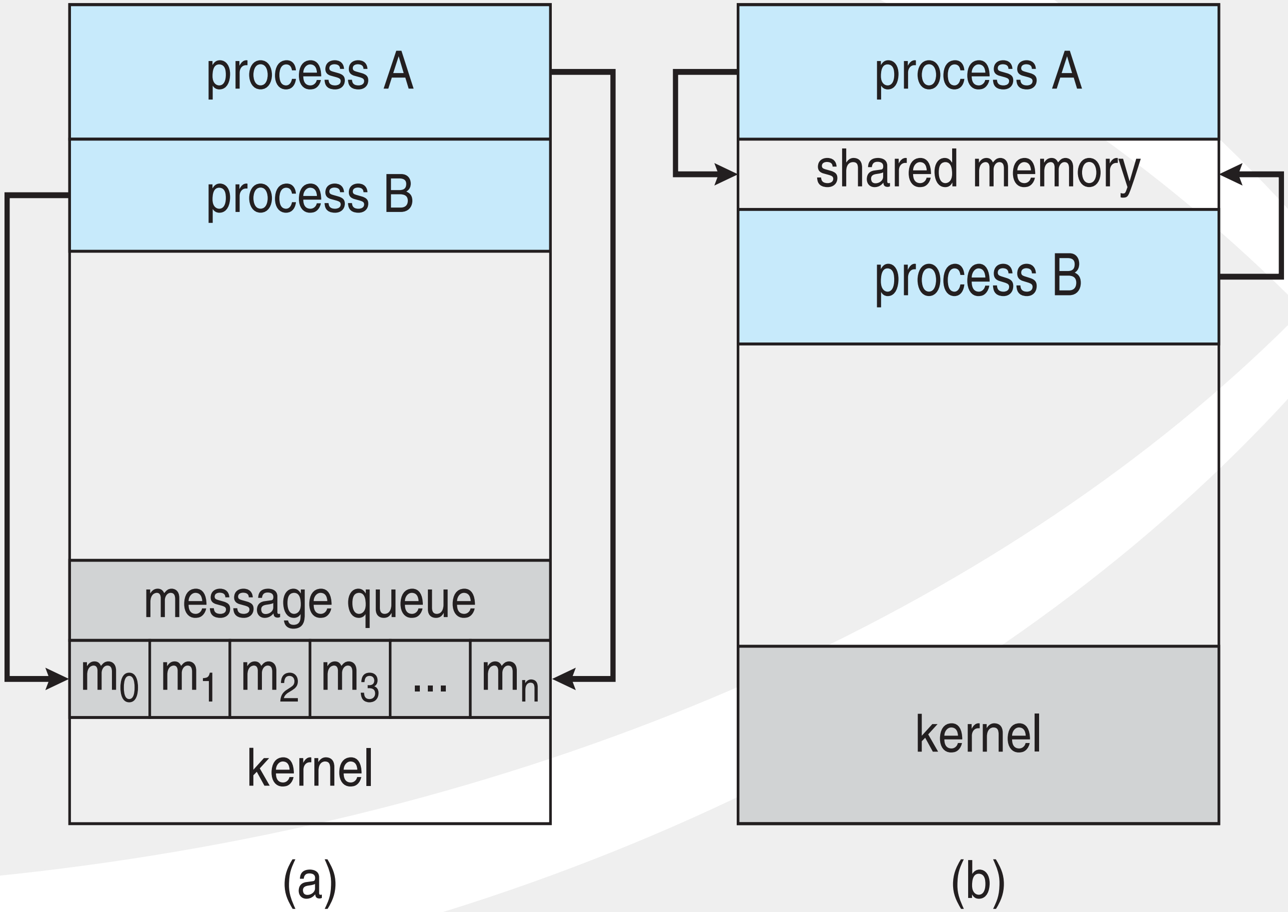


Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



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

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

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 (Cont.)

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