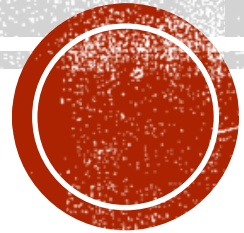
PROCESS MANAGEMENT

Chapter2.1_Process_Management



2.1 INTRODUCTION TO PROCESSES:

- The process model
- Implementation of processes
- Threads
- Thread model
- Thread usage
- Implementing thread in user space

WHAT IS A PROCESS?

- The program in execution. **ACTIVE**
- A process is different than a program. **PASSIVE**
- *A process is an activity of some kind, it has program, input, output and state.*

- **Pseudoparallelism/Multiprogramming:**
  - Fast switching of CPU executing multiple jobs within seconds
- **Multiprocessor:**
  - Two or more CPU sharing the same physical memory

TWO ANALOGIES TO UNDERSTAND PROCESS 1/2

- ***Scenario-1:*** *A computer scientist is baking a birthday cake for his daughter*

Computer scientist - CPU
recipe – program (method to cook – algorithm)
cake Ingredients - input data

Activities are processes
- reading the recipe
- fetching the ingredients
- backing the cake

TWO ANALOGIES TO UNDERSTAND PROCESS 2/2

- ***Scenario-2:*** *Scientist's son comes running in crying, saying he has been stung by a bee.*

Scientist records where he was in the recipe - the state of running process saved.

Reach first aid book and materials - Another process fetched

Follow the first aid action - Processor switched for new (high priority job) process

On completion of aid - Completion of high priority

baking starts again from the last instruction where it was left

THE PROCESS MODEL

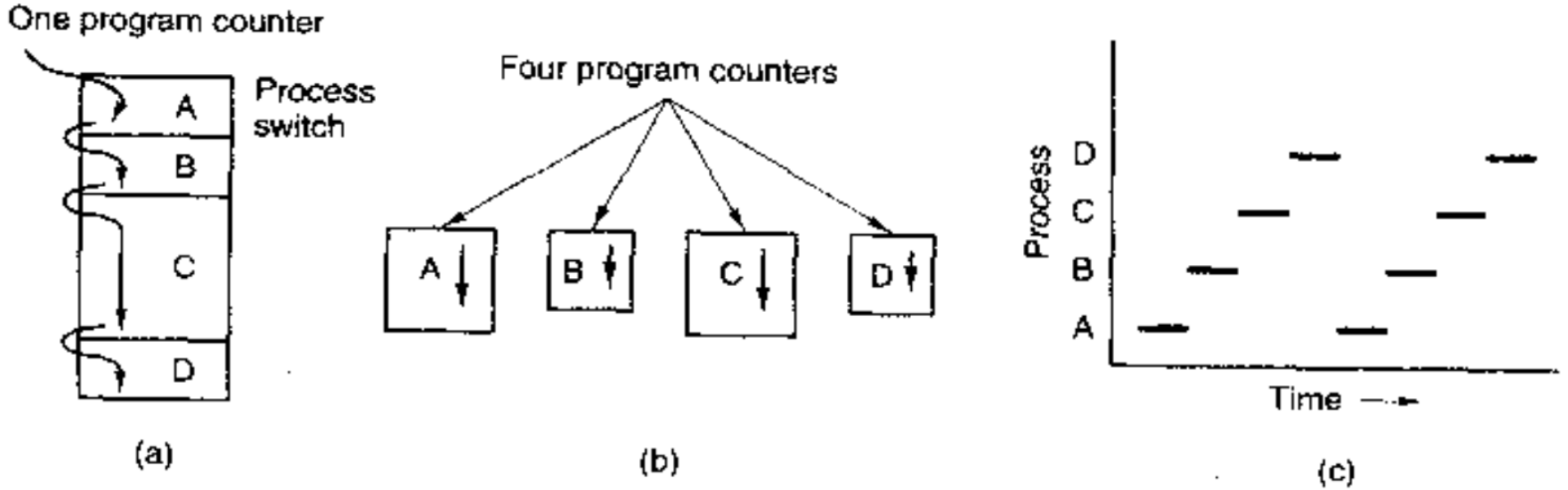
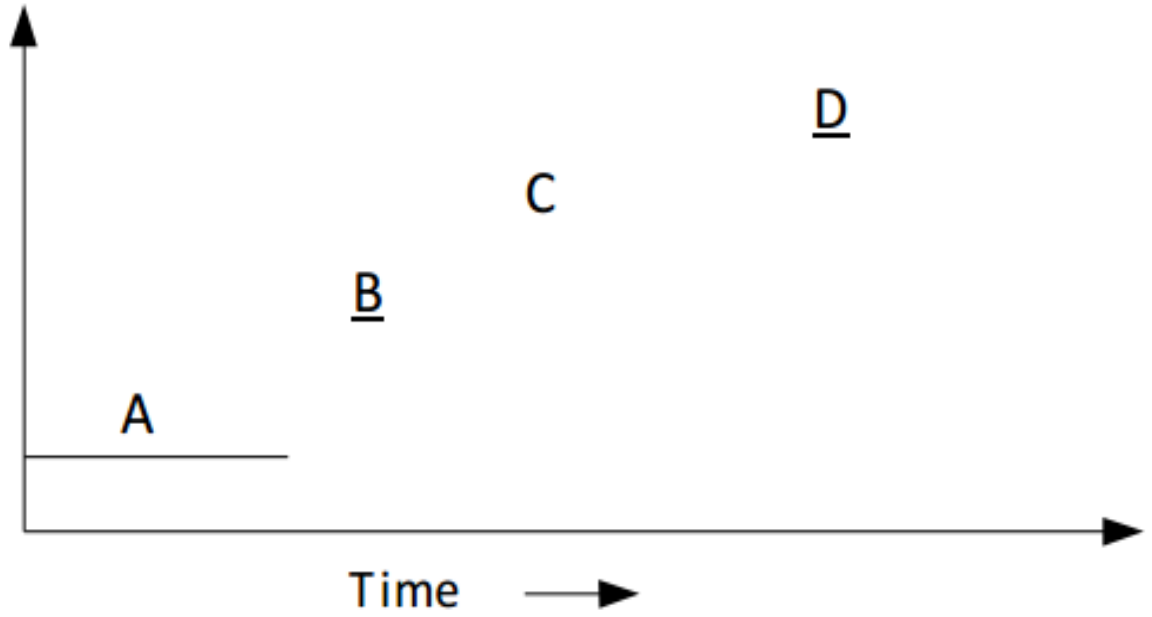


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

THE PROCESS MODEL

- Uniprogramming
- Multiprogramming
- Multiprocessing

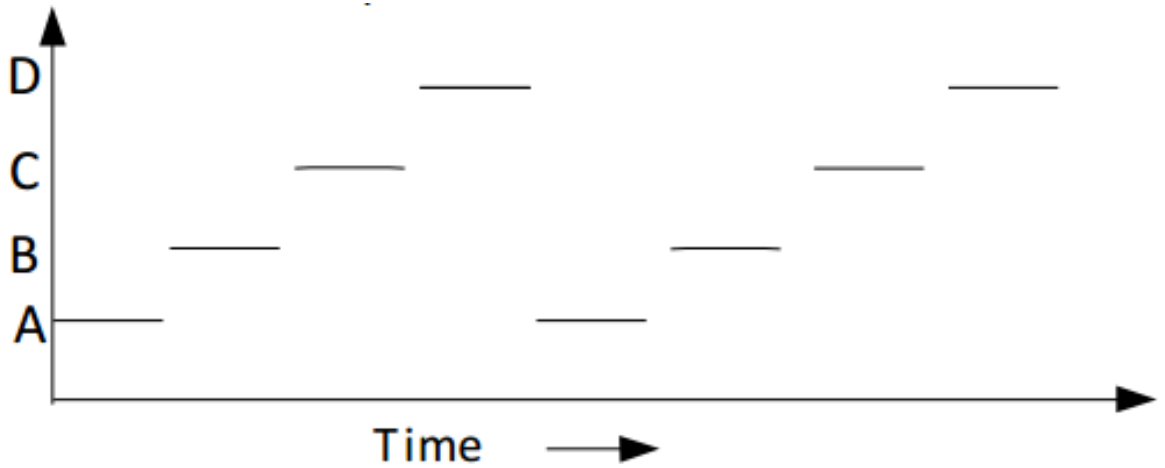
UNIPROGRAMMING



- Only one process at a time.
- Examples: Older systems
- Advantages: Easier for OS designer
- Disadvantages: Not convenient for user and poor performance

MULTIPROGRAMMING

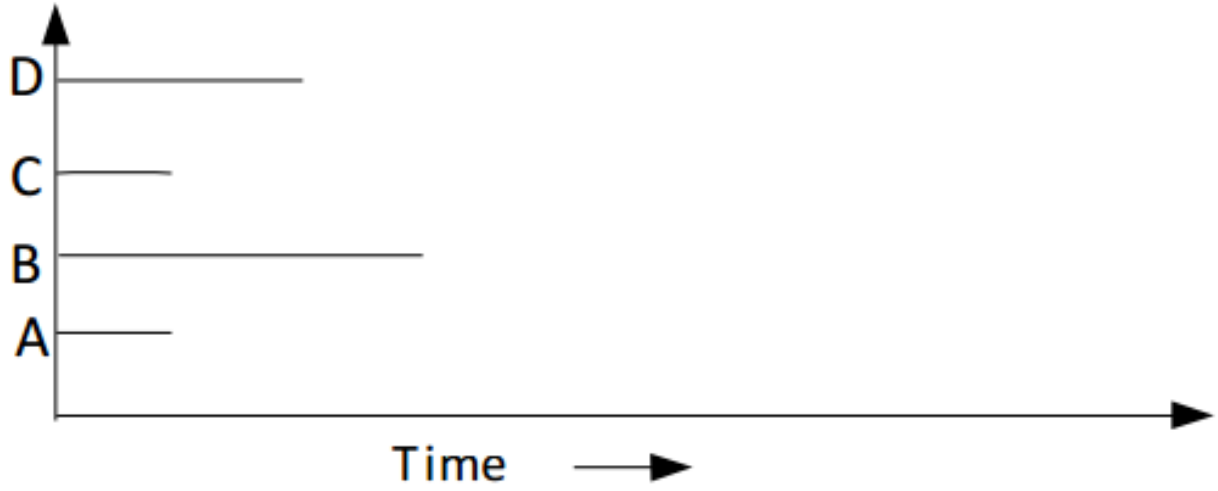
- *Multiple processes at a time.*



- OS requirements for multiprogramming:
  - ***Policy:*** to determine which process is to schedule.
  - ***Mechanism:*** to switch between the processes.
- Examples: Unix, WindowsNT
- Advantages: Better system performance and user convenience.
- Disadvantages: Complexity in OS

MULTIPROCESSING

- System with multiple processors



PROCESS STATE

- NEW: A process being created
- Running: Instructions are being executed.
- Waiting: The processes is waiting for some event to occur(such as I/O completion or reception of signal.
- Ready: The process is waiting to be assigned to a processor.
- Terminated: the processes has finished execution.

PROCESS STATE

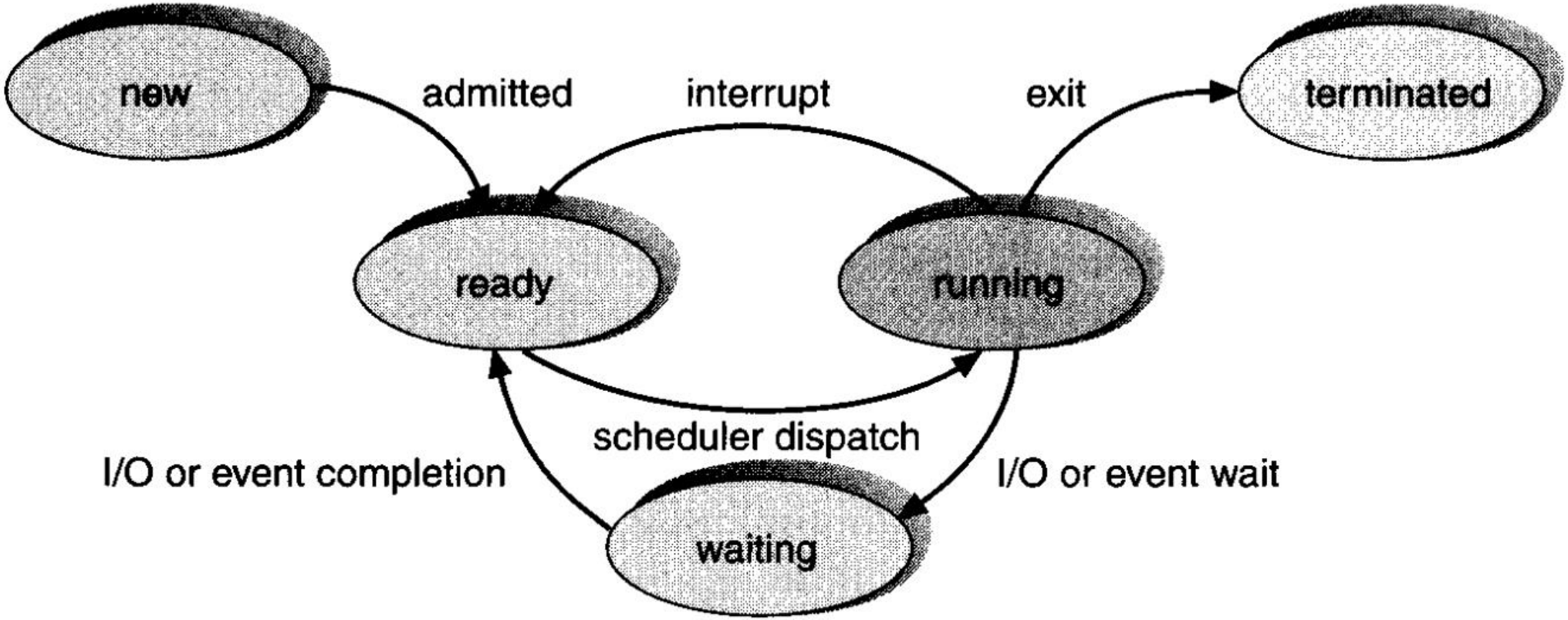
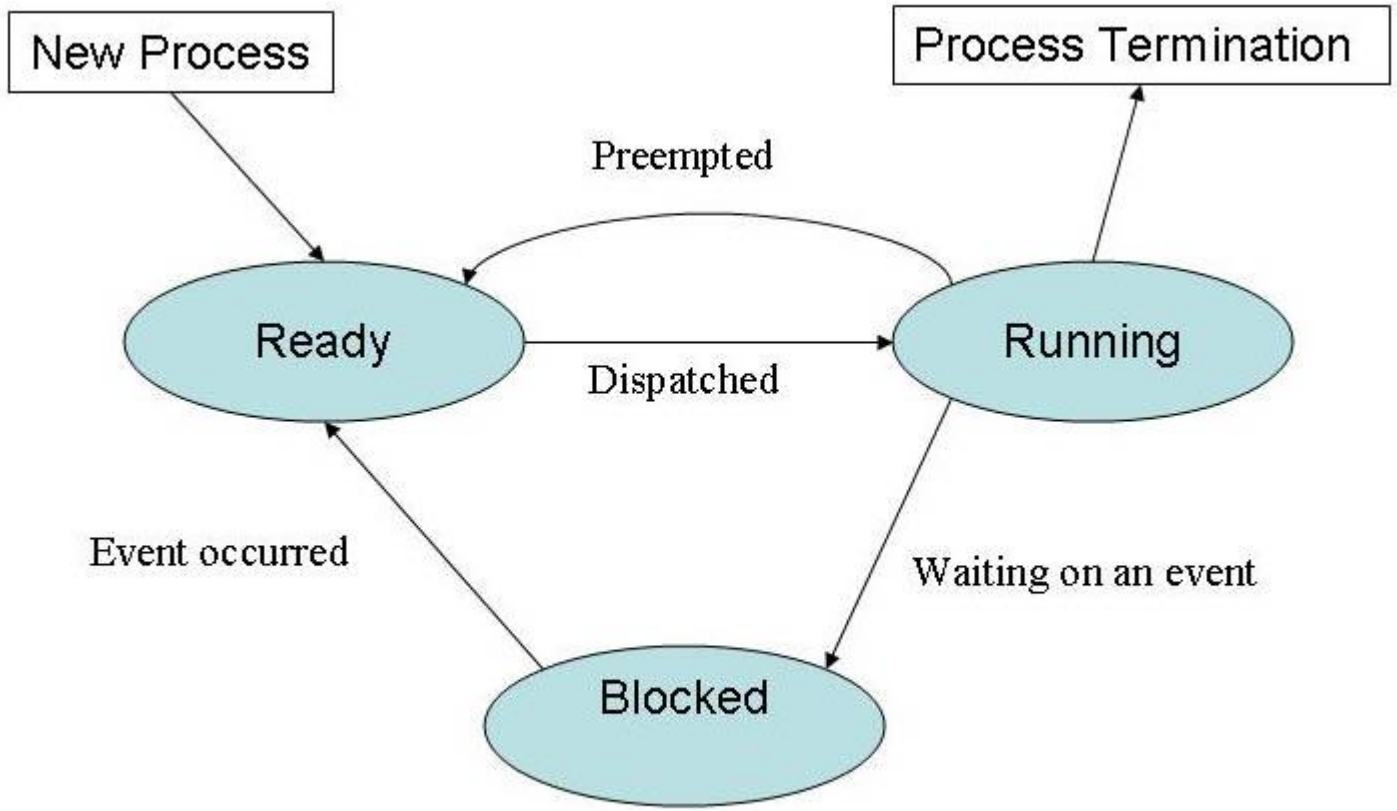


Figure 4.1 Diagram of process state.

OPERATIONS ON PROCESS

- Process Creation
- Process Preemption
- Process Blocking
- Process Termination



PROCESS CREATION

- 4 Principal events that cause the processes to be created:

- 1. A user request to create a new process
- 2. System Initialization
- 3. Execution of a process creation system call by a running process
- 4. Initiation of a batch job

2 WAYS TO CREATE PROCESS

- Build a new one from scratch

  - Load specified code and data into memory.
  - Create and initialize PCB.
  - Put processes on the ready list.

- Clone an existing one (e.g. Unix fork() syscall)

  - Stop the current process and save its state.
  - Make copy of code, data, stack, and PCB.
  - Add new process PCB to ready list.

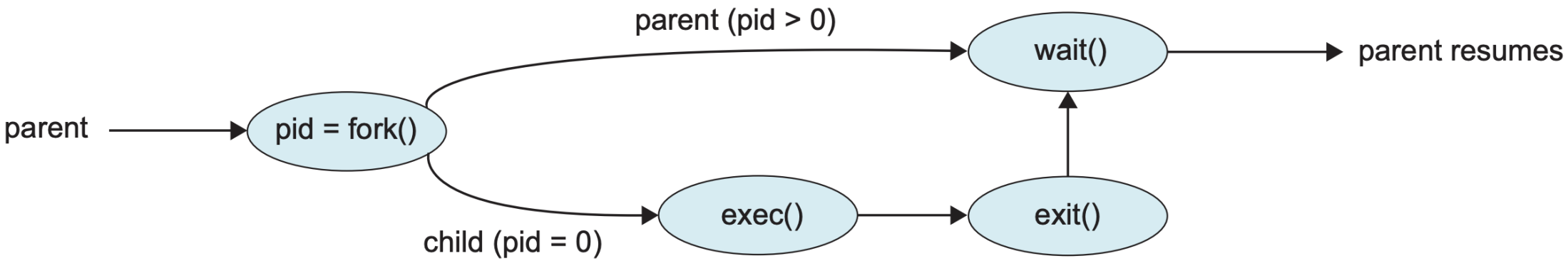


Figure 3.10 Process creation using the `fork()` system call.

CREATING PROCESS

```
#include<stdio.h>

int main(int argc, char *argv[]){
        int pid;
        pid = fork();                           /* create new process */
        if(pid < 0){                            /* error occurred */
                fprintf(stderr, "fork failed");
                exit(-1);
        } else if(pid == 0){                    /* child process */
                execlp("/bin/ls", "ls", Null);
        }else{                                  /* parent process */
                wait(Null);
                printf("Child Complete");
                exit(0);
        }
                return 0;
}
```

PROCESS TERMINATION

- Process terminates because of the following conditions:
  - Normal Exit(voluntary)
    - Screen based : close button
  - Error Exit (voluntary)
    - Caused by Program bug (referencing Non existent memory, divide by zero)
    - Some systems may try to handle such errors by themself
  - Fatal Error (involuntary)
    - An illegal instructions has been attempted
    - Invalid data or code has been accessed
    - An operation is not allowed in the current CPU mode
  - Killed by another process(involuntary)

PROCESS CONTROL BLOCK (PCB)

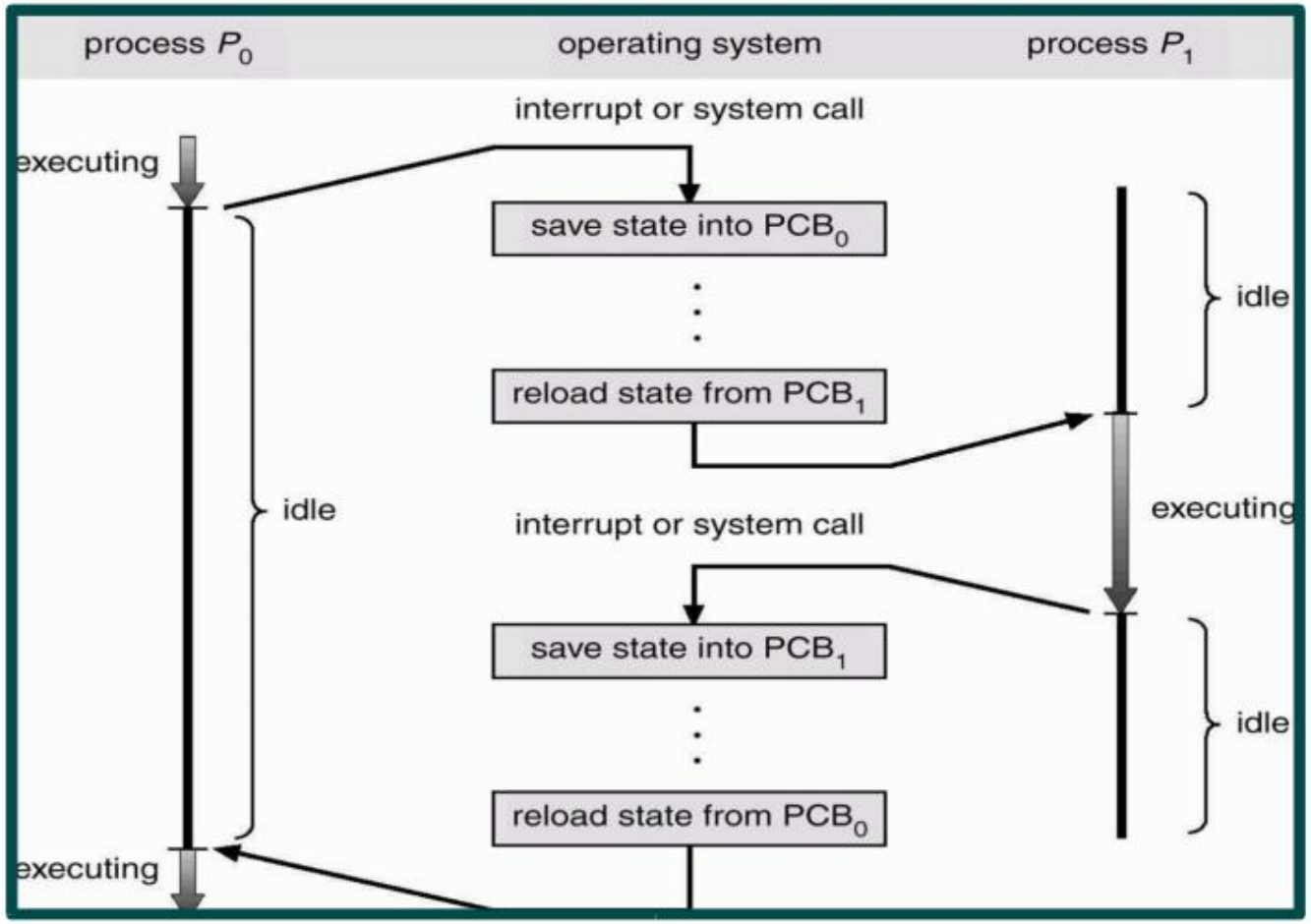
- Process must be saved when the process is switched from one state to another so that it can be restarted later as it had never been stopped.
- *The **PCB** is the data structure containing certain important information about the process -also called process table or processor descriptor.*

  - ***Process state*:** running, ready, blocked.
  - ***Program counter*:** Address of next instruction for the process.
  - ***Registers*:** Stack pointer, accumulator, PSW etc.
  - ***Scheduling information*:** Process priority, pointer to scheduling queue etc.
  - ***Memory-allocation*:** value of base and limit register, page table, segment table etc.
  - ***Accounting information*:** time limit, process numbers etc.
  - ***Status information:*** list of I/O devices, list of open files etc.

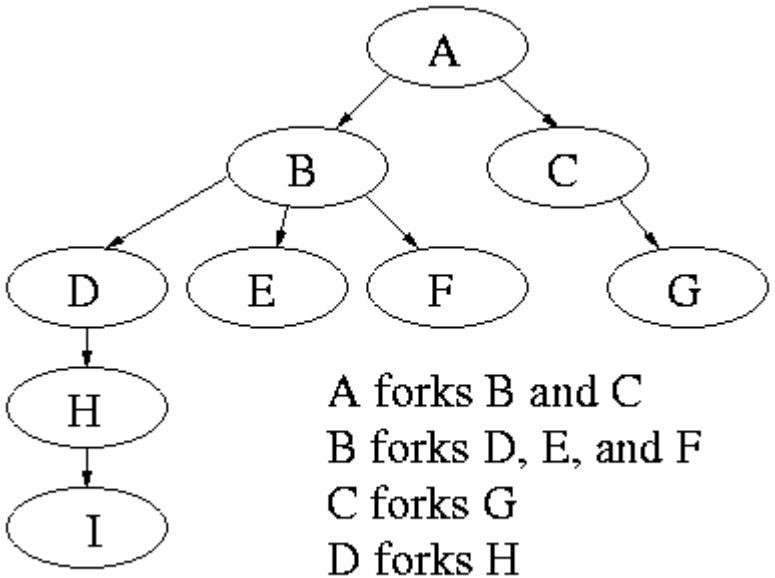
TYPICAL PROCESS TABLE ENTRIES

Process Management	Memory Management	File Management
Registers	Pointers to text segment info	Root directory
Program Counter	Pointers to data segment info	Working directory
Program Status Word	Pointers to stack segment info	File descriptors
Stack Pointer		User ID
Process State		Group ID
Priority		
Scheduling Parameters		
Process ID		
Parent Process		
Process Group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of Next Alarm		

PCB(PROCESS CONTROL BLOCK)



PROCESS HIERARCHIES



A forks B and C
B forks D, E, and F
C forks G
D forks H
H forks I

COOPERATING PROCESS

- Modularity
  - Dividing complicated task into smaller subtask
- Information Sharing
- Convenience
  - Easy if we divide task such as compiling, printing, editing etc
- Computation Speedup

METHODS OF COOPERATION

- Co-operation by Sharing
  - Memory, variables,files,databases, etc
  - May involve Race Condition
- Co-operation by communication
  - Using messages
  - My involve deadlock and starvation

THREAD

- Threads are like Process.
- Allows more than one job to be performed at a time.
- Thread – **Lightweight** Process. Process - **Heavyweight** Process.
- Many applications now-a-days are multithreaded
  - Web Browser
    - A Thread for retrieving data from network
    - Another Thread for displaying text and images
  - Word Processor
    - A Thread for displaying graphics
    - Another thread for reading keystokes from user
    - Third Thread for performing grammar checks in background

SINGLE VS MULTITHREADED PROCESS

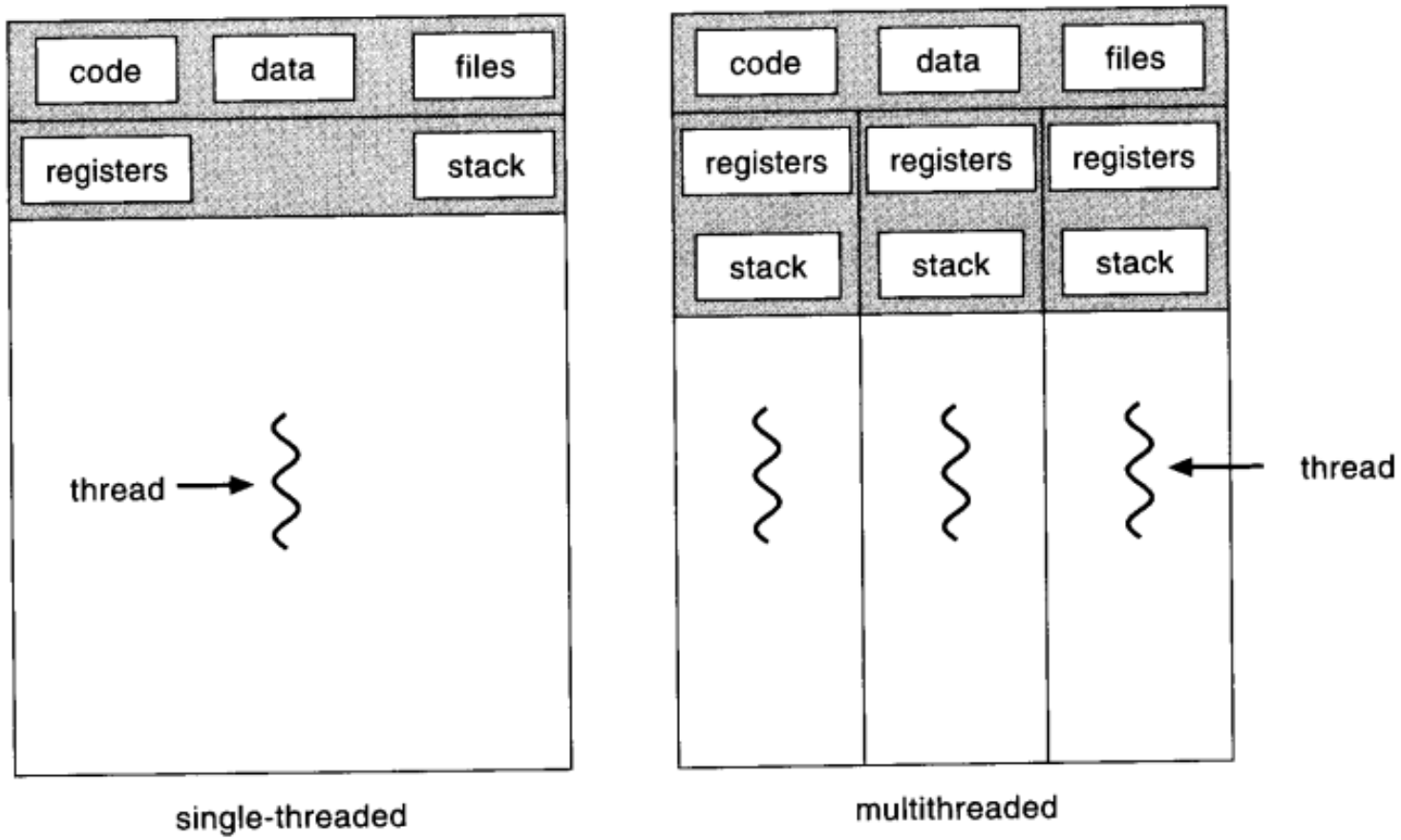
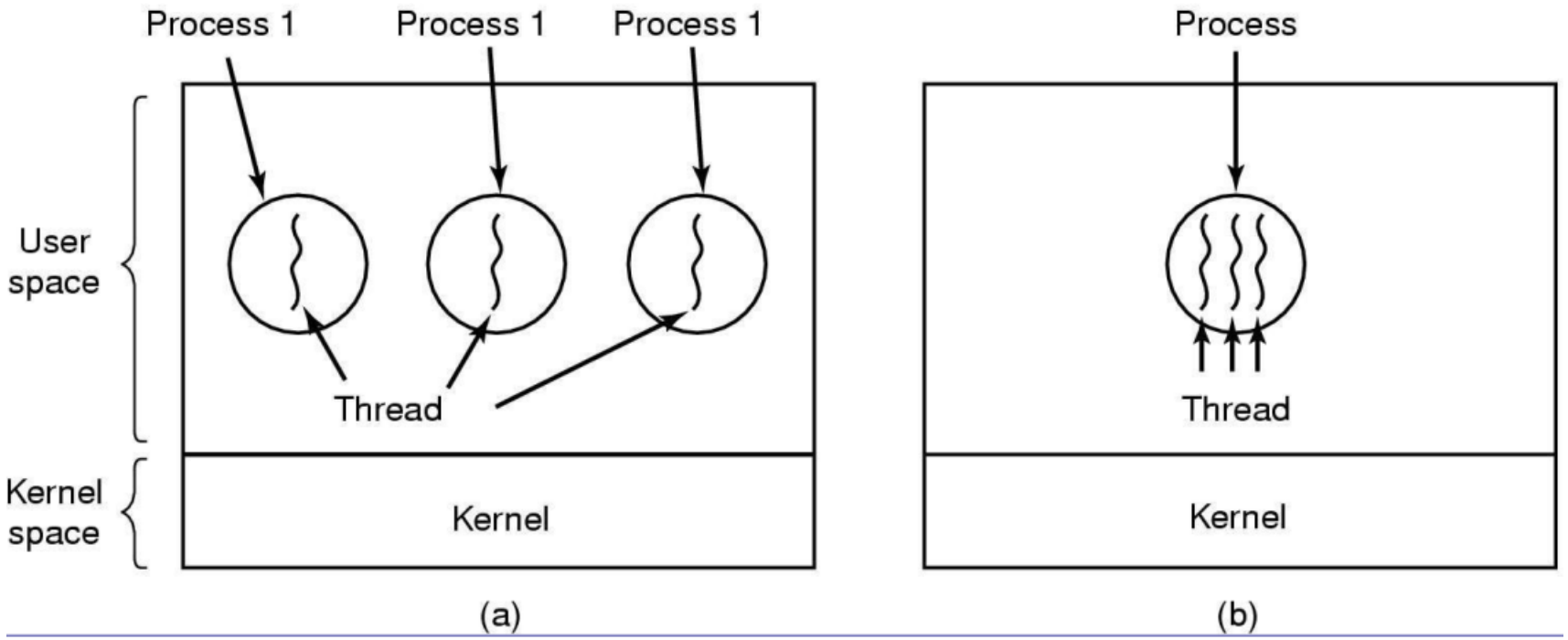


Figure 5.1 Single- and multithreaded processes.

PROCESS VS THREAD



(a) Three process each with one thread (b) one process with three threads.

a) Useful when three processes are unrelated
b) Appropriate if three threads are actually part of same job and are actively cooperating with each other.

BENEFITS OF THREADING

- 1. Responsiveness
  - A process can still interact with user while performing heavy task in background
- 2. Resource sharing
  - Several different threads can share same address space with its process
- 3. Economy
  - Allocating memory and resource is costly.
  - Much more time consuming to create process than thread.
  - Multiple threads share same memory and resources.
- 4. Utilization of multiprocessor architecture
  - Multithreading may be useful to run different tasks in parallel

USER THREADS VS. KERNEL THREAD

- User Threads:
  - Thread management done on user-level by using thread library
  - Library provides support for thread creation, scheduling and management
  - No kernel intervention required
  - Fast to create
  - If kernel is single threaded, blocking system calls will cause the entire process to block
    - Example: POSIX (Pthreads), Mach(C-threads), Solaris 2 (UI-Threads)
- Kernel Threads:
  - Thread Management done by OS
  - Kernel provides support for thread creation, scheduling and management
  - Slower to create and manage
  - Blocking system calls are no problem
  - Most OS support it
    - WinX, Linux.

MULTI-THREADING MODELS:

- One-to-One Model
- Many-to-One Model
- Many-to-Many Model

ONE-TO-ONE MODEL

- One kernel thread for each user thread
- is a very widespread model
  - Linux
  - sometimes referred to as "native threads."
- **Pros:**
- Threads can execute on different CPUs
- Threads do not block each other
- Shared memory
- **Cons:**
- Setup overhead
- Linux kernel bug with lots of threads
- Only low number of threads can be created

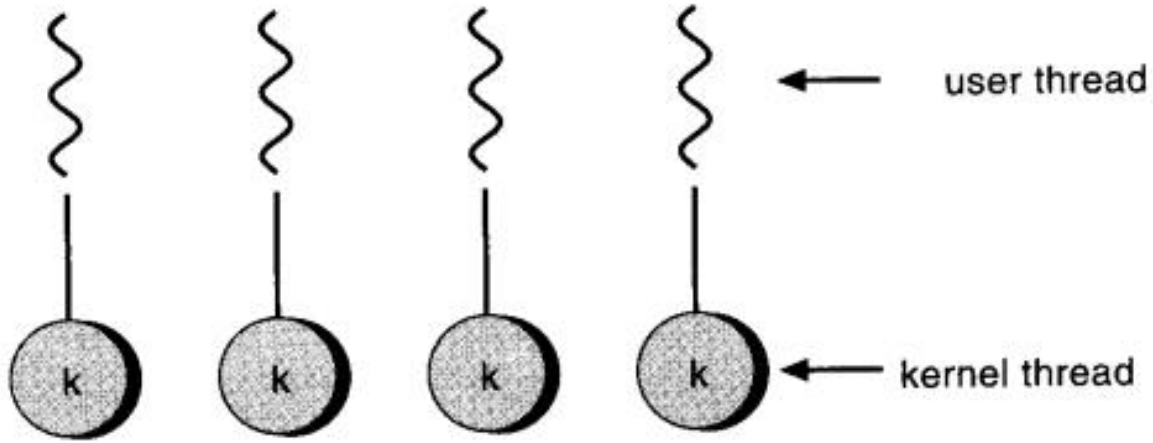


Figure 5.3 One-to-one model.

MANY-TO-ONE MODEL

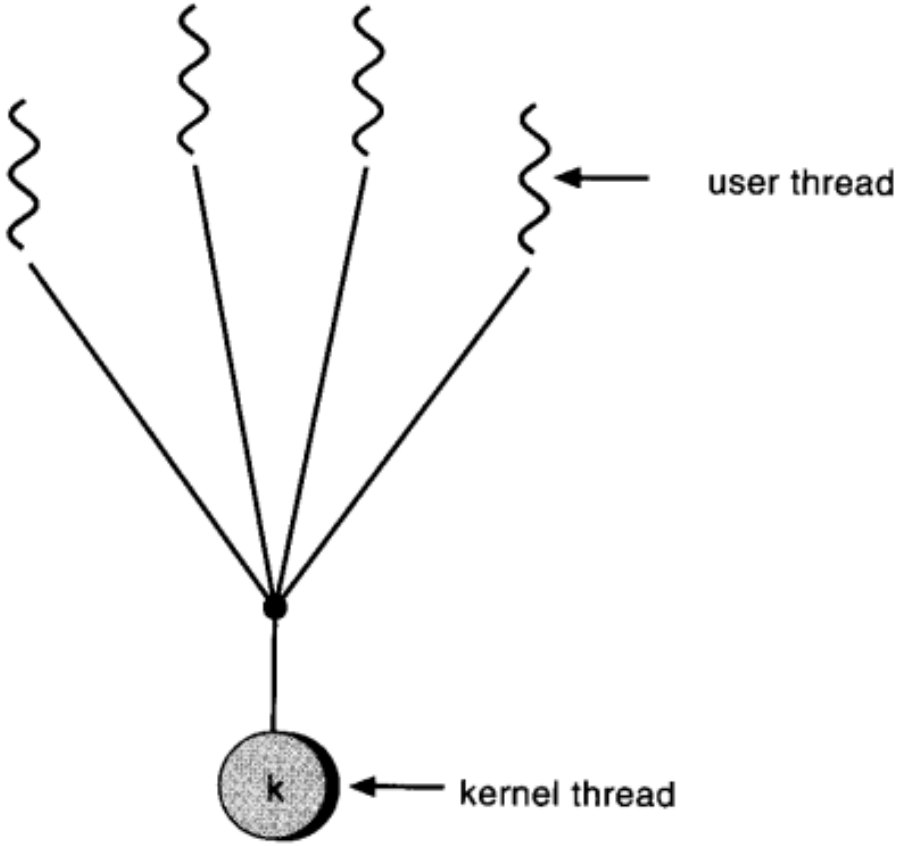


Figure 5.2 Many-to-one model.

- one kernel thread for N user threads
  - commonly called "green threads" or "lightweight threads."
- **Pros:**
- Thread creation, execution, and cleanup are cheap
- Lots of threads can be created (10s of thousands or more)
- **Cons:**
- Kernel scheduler doesn't know about threads so they can't be scheduled across CPUs or take advantage of SMP(Symmetric Processing)
- Blocking I/O operations can block all the green threads

MANY-TO-MANY MODEL

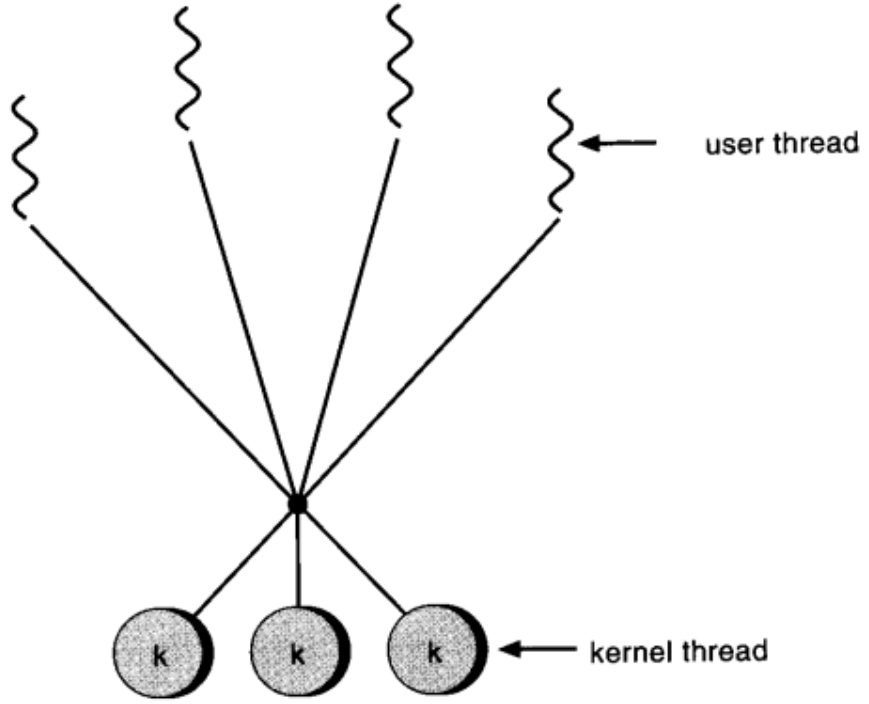


Figure 5.4 Many-to-many model.

M kernel threads for N user threads,

hybrid of the previous two models.

- **Pros:**
- Take advantage of multiple CPUs
- Not all threads are blocked by blocking system calls
- Cheap creation, execution, and cleanup
- **Cons:**
- Need scheduler in userSpace and kernel to work with each other
- Green threads doing blocking I/O operations will block all other green threads sharing same kernel thread
- Difficult to write, maintain, and debug code