CAMBRIDGE
INSTITUTE OF TECHNOLOGY

Process, Threads and Process Synchronization

Department of
Computer Science & Engineering

www.cambridge.edu.in

CAMBRIDGE
INSTITUTE OF TECHNOLOGY

Process, Threads and Process Synchronization

topics

Process and Threads: Processes and programs, Implementing process, case study: Threads, processes in UNIX.

Process Synchronization: Race conditions, Critical section, control synchronization and invisible operations, synchronization approaches, structure of concurrent systems, classic approach synchronization problems, semaphores, monitors, case study: process synchronization in UNIX.

Introduction

- Processes and Programs
- Implementing Processes
- Threads
- Case Studies of Processes and Threads

Processes and Programs

- What Is a Process?
- Relationships between Processes and Programs
- Child Processes
- Concurrency and Parallelism

What Is a Process?

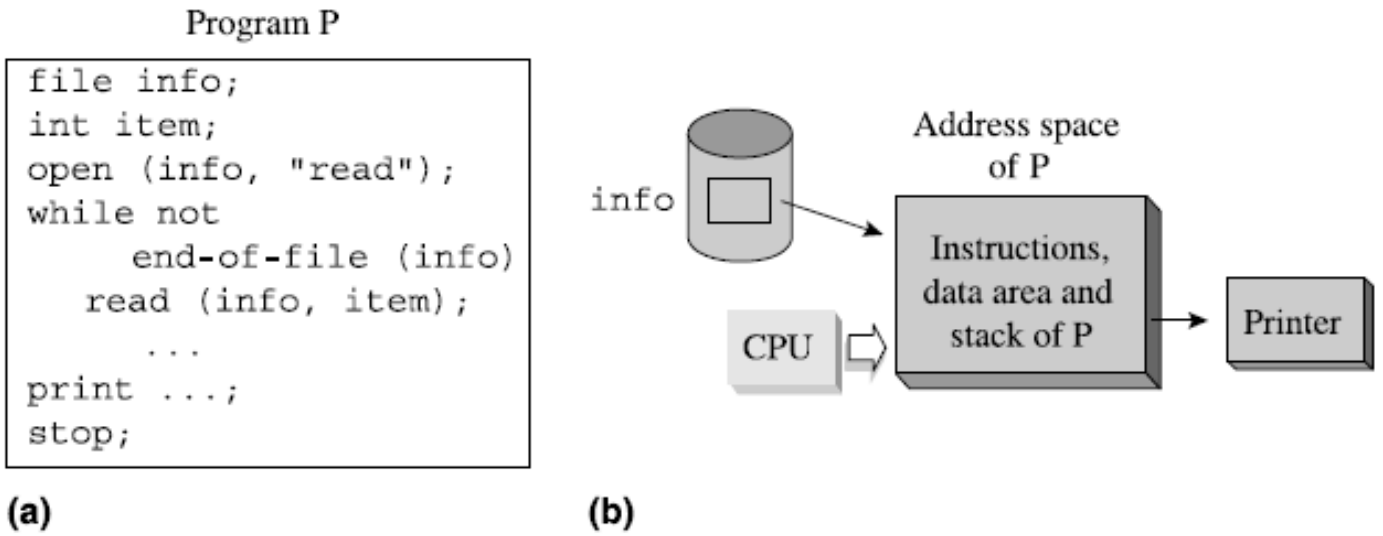

Figure 5.1 A program and an abstract view of its execution.

Definition 5.1 Process An execution of a program using resources allocated to it.

- A process comprises six components:
  - *(id, code, data, stack, resources, CPU state)*

Relationships between Processes and Programs

- A program is a set of functions and procedures
  - Functions may be separate processes, or they may constitute the code part of a single process

Table 5.1 Relationships between Processes and Programs

Relationship	Examples
One-to-one	A single execution of a sequential program.
Many-to-one	Many simultaneous executions of a program, execution of a concurrent program.

Child Processes

- Kernel initiates an execution of a program by creating a process for it
  - Primary process may make system calls to create other processes
    - *Child processes* and *parents* create a *process tree*
- Typically, a process creates one or more child processes and delegates some of its work to each
  - *Multitasking* within an application

Child Processes (continued)

Table 5.2 Benefits of Child Processes

Benefit	Explanation
Computation speedup	Actions that the primary process of an application would have performed sequentially if it did not create child processes, would be performed concurrently when it creates child processes. It may reduce the duration, i.e., running time, of the application.
Priority for critical functions	A child process that performs a critical function may be assigned a high priority; it may help to meet the real-time requirements of an application.
Guarding a parent process against errors	The kernel aborts a child process if an error arises during its operation. The parent process is not affected by the error; it may be able to perform a recovery action.

Example: Child Processes in a Real-Time Application

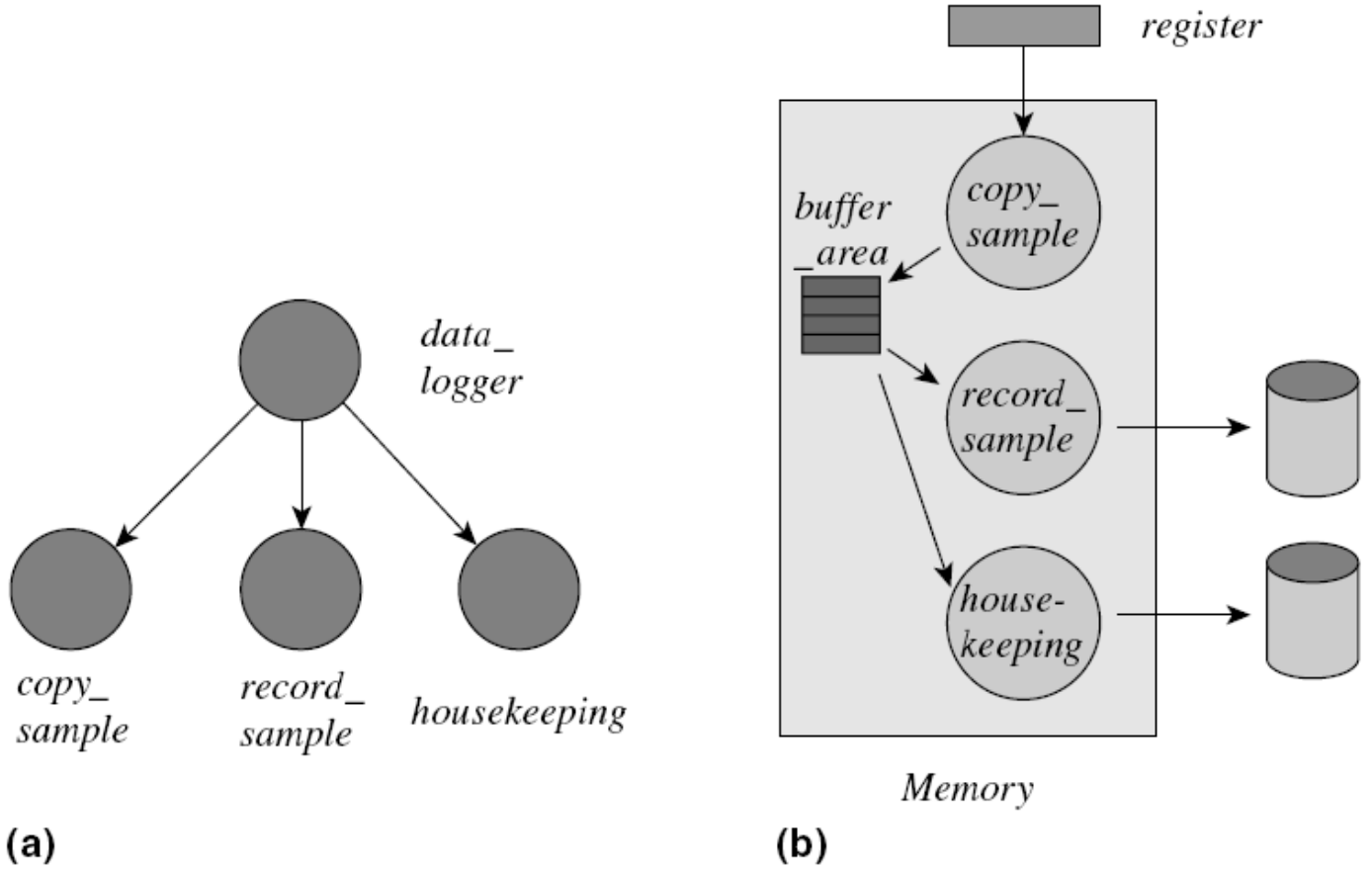

Figure 5.2 Real-time application of Section 3.7: (a) process tree; (b) processes.

Concurrency and Parallelism

- *Parallelism:* quality of occurring at the same time
  - Two tasks are parallel if they are performed at the same time
  - Obtained by using multiple CPUs
    - As in a multiprocessor system
- *Concurrency* is an illusion of parallelism
  - Two tasks are concurrent if there is an illusion that they are being performed in parallel whereas only one of them may be performed at any time
  - In an OS, obtained by interleaving operation of processes on the CPU
- Both concurrency and parallelism can provide better throughput

Implementing Processes

- To OS, a process is a unit of computational work
  - Kernel's primary task is to control operation of processes to provide effective utilization of the computer system

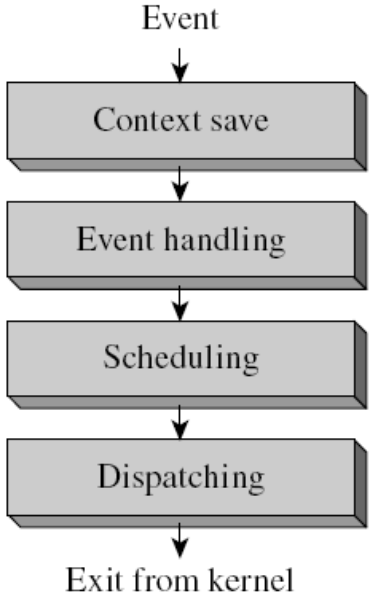

Figure 5.3 Fundamental functions of the kernel for controlling processes.

Process States and State Transitions

Definition 5.2 Process state The indicator that describes the nature of the current activity of a process.

Table 5.3 Fundamental Process States

State	Description
Running	A CPU is currently executing instructions in the process code.
Blocked	The process has to wait until a resource request made by it is granted, or it wishes to wait until a specific event occurs.
Ready	The process wishes to use the CPU to continue its operation; however, it has not been dispatched.
Terminated	The operation of the process, i.e., the execution of the program represented by it, has completed normally, or the OS has aborted it.

Process States and State Transitions (continued)

- A *state transition* for a process is a change in its state
  - Caused by the occurrence of some event such as the start or end of an I/O operation

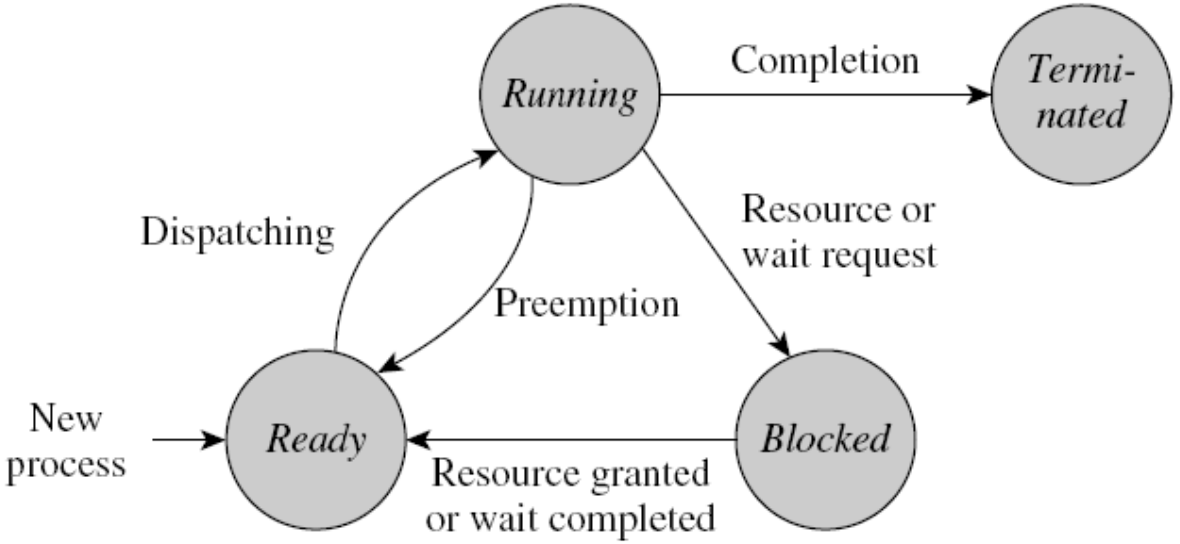

Figure 5.4 Fundamental state transitions for a process.

Process States and State Transitions (continued)

Table 5.4 Causes of Fundamental State Transitions for a Process

State transition	Description
ready → *running*	The process is dispatched. The CPU begins or resumes execution of its instructions.
blocked → *ready*	A request made by the process is granted or an event for which it was waiting occurs.
running → *ready*	The process is preempted because the kernel decides to schedule some other process. This transition occurs either because a higher-priority process becomes *ready*, or because the time slice of the process elapses.
running → *blocked*	The process in operation makes a system call to indicate that it wishes to wait until some resource request made by it is granted, or until a specific event occurs in the system. Five major causes of blocking are: • Process requests an I/O operation • Process requests a resource • Process wishes to wait for a specified interval of time • Process waits for a message from another process • Process waits for some action by another process.

Table 5.4 Causes of Fundamental State Transitions for a Process

State transition	Description
running → *terminated*	Execution of the program is completed. Five primary reasons for process termination are: • *Self-termination:* The process in operation either completes its task or realizes that it cannot operate meaningfully and makes a "terminate me" system call. Examples of the latter condition are incorrect or inconsistent data, or inability to access data in a desired manner, e.g., incorrect file access privileges. • *Termination by a parent:* A process makes a "terminate P_i" system call to terminate a child process P_i, when it finds that execution of the child process is no longer necessary or meaningful. • *Exceeding resource utilization:* An OS may limit the resources that a process may consume. A process exceeding a resource limit would be aborted by the kernel. • *Abnormal conditions during operation:* The kernel aborts a process if an abnormal condition arises due to the instruction being executed, e.g., execution of an invalid instruction, execution of a privileged instruction, arithmetic conditions like overflow, or memory protection violation. • *Incorrect interaction with other processes:* The kernel may abort a process if it gets involved in a deadlock.

Example: Process State Transitions

- A system contains two processes P_1 and P_2

Table 5.5 Process State Transitions in a Time-Sharing System

Time	Event	Remarks	New states	
			P_1	P_2
0		P_1 is scheduled	*running*	*ready*
10	P_1 is preempted	P_2 is scheduled	*ready*	*running*
20	P_2 is preempted	P_1 is scheduled	*running*	*ready*
25	P_1 starts I/O	P_2 is scheduled	*blocked*	*running*
35	P_2 is preempted	—	*blocked*	*ready*
		P_2 is scheduled	*blocked*	*running*
45	P_2 starts I/O	—	*blocked*	*blocked*

Suspended Processes

- A kernel needs additional states to describe processes suspended due to swapping

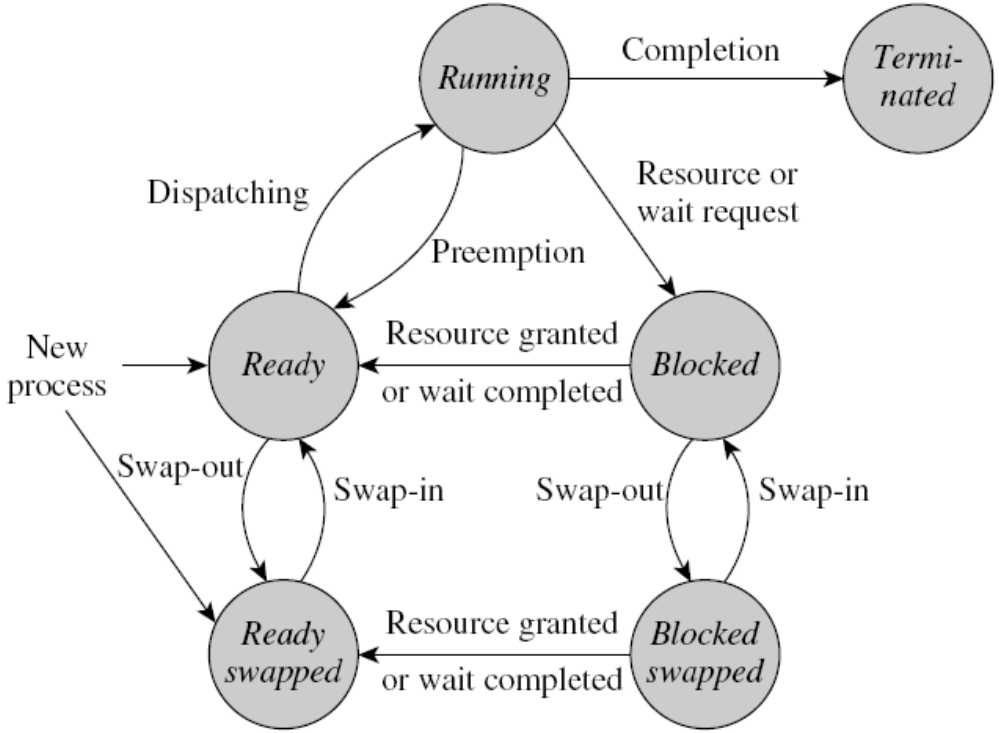

Figure 5.5 Process states and state transitions using two swapped states.

Process Context and the Process Control Block

- Kernel allocates resources to a process and schedules it for use of the CPU
  - The kernel's view of a process is comprised of the process context and the process control block

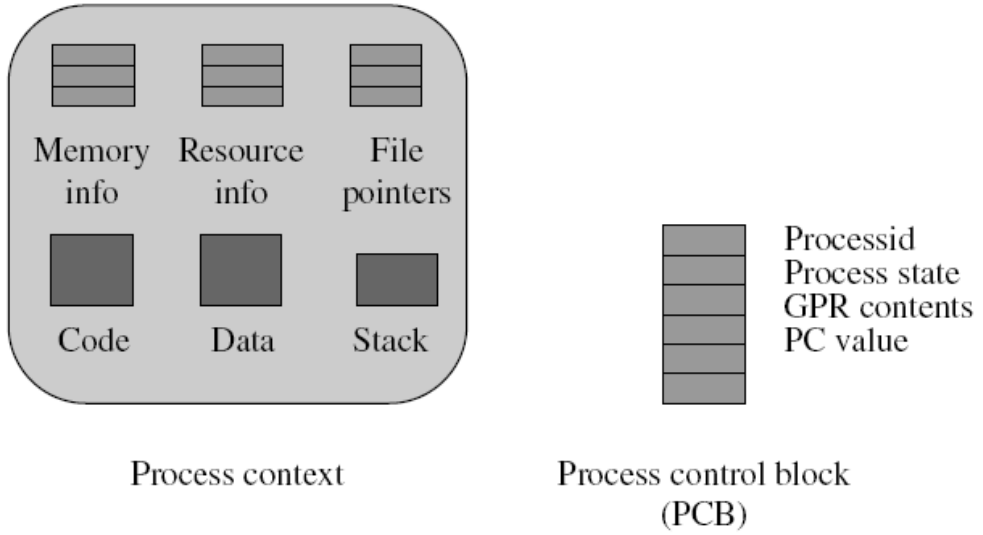

Figure 5.6 Kernel's view of a process.

Table 5.6 Fields of the Process Control Block (PCB)

PCB field	Contents
Process id	The unique id assigned to the process at its creation.
Parent, child ids	These ids are used for process synchronization, typically for a process to check if a child process has terminated.
Priority	The priority is typically a numeric value. A process is assigned a priority at its creation. The kernel may change the priority dynamically depending on the nature of the process (whether CPU-bound or I/O-bound), its age, and the resources consumed by it (typically CPU time).
Process state	The current state of the process.
PSW	This is a snapshot, i.e., an image, of the PSW when the process last got blocked or was preempted. Loading this snapshot back into the PSW would resume operation of the process. (See Fig. 2.2 for fields of the PSW.)
GPRs	Contents of the general-purpose registers when the process last got blocked or was preempted.
Event information	For a process in the *blocked* state, this field contains information concerning the event for which the process is waiting.
Signal information	Information concerning locations of signal handlers (see Section 5.2.6).
PCB pointer	This field is used to form a list of PCBs for scheduling purposes.

Context Save, Scheduling, and Dispatching

- Context save function:
  - Saves CPU state in PCB, and saves information concerning context
  - Changes process state from *running* to *ready*
- Scheduling function:
  - Uses process state information from PCBs to select a *ready* process for execution and passes its id to dispatching function
- Dispatching function:
  - Sets up context of process, changes its state to *running*, and loads saved CPU state from PCB into CPU
  - Flushes address translation buffers used by MMU

Event Handling

- Events that occur during the operation of an OS:
  1. Process creation event
  2. Process termination event
  3. Timer event
  4. Resource request event
  5. Resource release event
  6. I/O initiation request event

Event Handling (continued)

- Events that occur during the operation of an OS (continued):
  7. I/O completion event
  8. Message send event
  9. Message receive event
  10. Signal send event
  11. Signal receive event
  12. A program interrupt
  13. A hardware malfunction event

Event Handling (continued)

- When an event occurs, the kernel must find the process whose state is affected by it
  - OSs use various schemes to speed this up
    - E.g., *event control blocks* (ECBs)

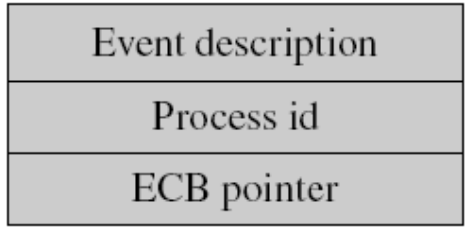

Figure 5.7 Event control block (ECB).

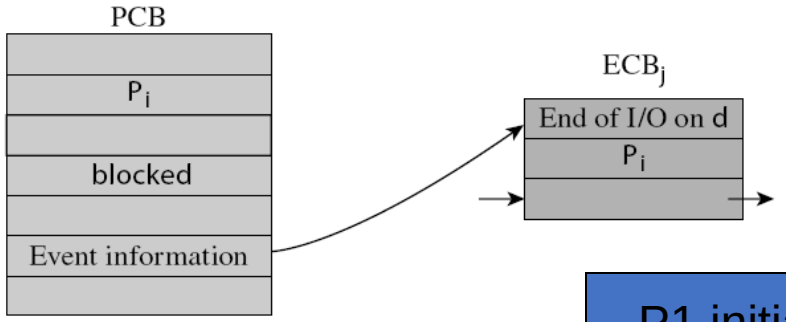

Figure 5.8 PCB-ECB interrelationship.

P1 initiates I/O operation on d

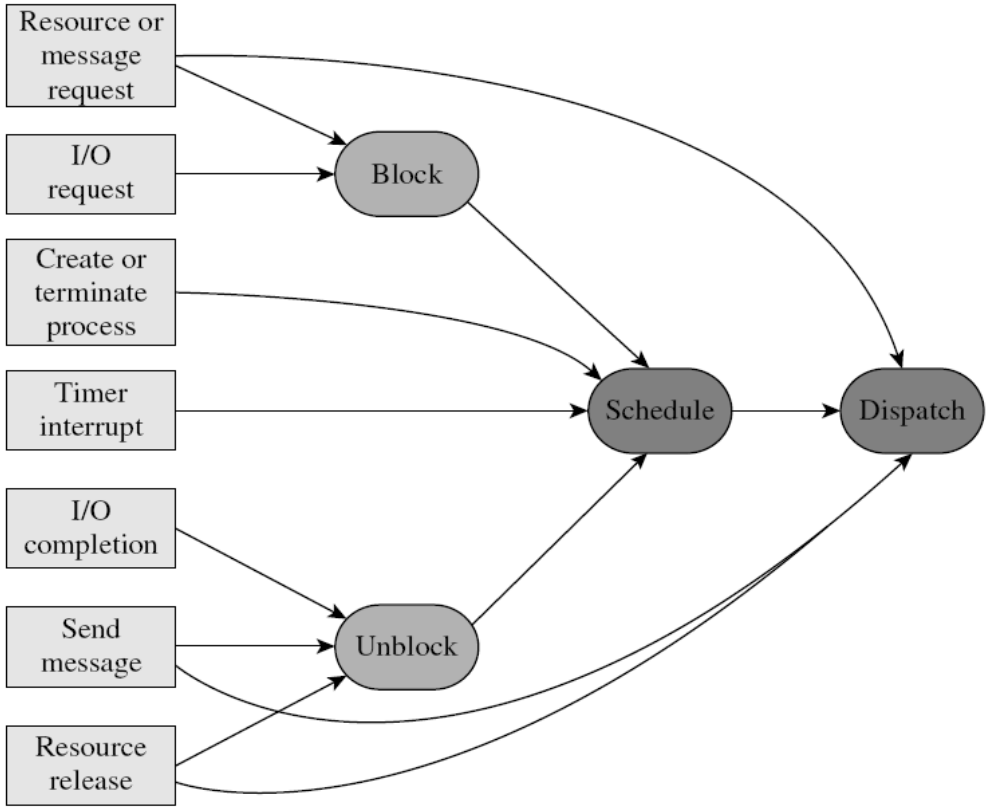

Figure 5.9 Event handling actions of the kernel.

Sharing, Communication and Synchronization Between Processes

Table 5.7 Four Kinds of Process Interaction

Kind of interaction	Description
Data sharing	Shared data may become inconsistent if several processes modify the data at the same time. Hence processes must interact to decide when it is safe for a process to modify or use shared data.
Message passing	Processes exchange information by sending messages to one another.
Synchronization	To fulfill a common goal, processes must coordinate their activities and perform their actions in a desired order.
Signals	A signal is used to convey occurrence of an exceptional situation to a process.

Signals

- A signal is used to notify an exceptional situation to a process and enable it to attend to it immediately
  - Situations and signal names/numbers defined in OS
    - CPU conditions like overflows
    - Conditions related to child processes
    - Resource utilization
    - Emergency communications from a user to a process
- Can be *synchronous* or *asynchronous*
- Handled by process-defined *signal handler* or OS provided *default handler*

Example: Signal handling

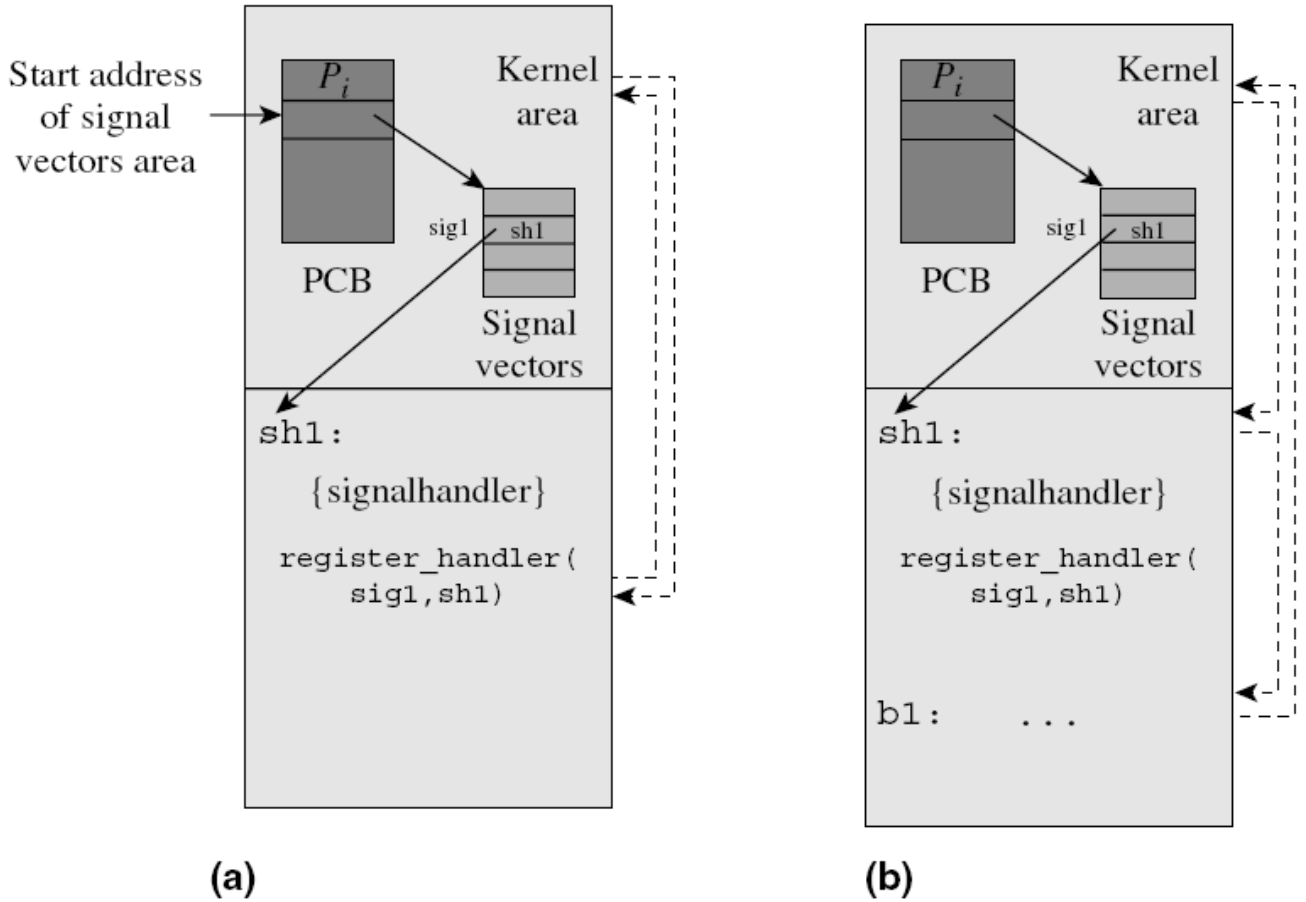

Figure 5.10 Signal handling by process P_i: (a) registering a signal handler; (b) invoking a signal handler.

Threads

Definition 5.3 Thread An execution of a program that uses the resources of a process.

- A thread is an alternative model of program execution
- A process creates a thread through a system call
- Thread operates within process context
- Use of threads effectively splits the process state into two parts
  - Resource state remains with process
  - CPU state is associated with thread
- Switching between threads incurs less overhead than switching between processes

Threads (continued)

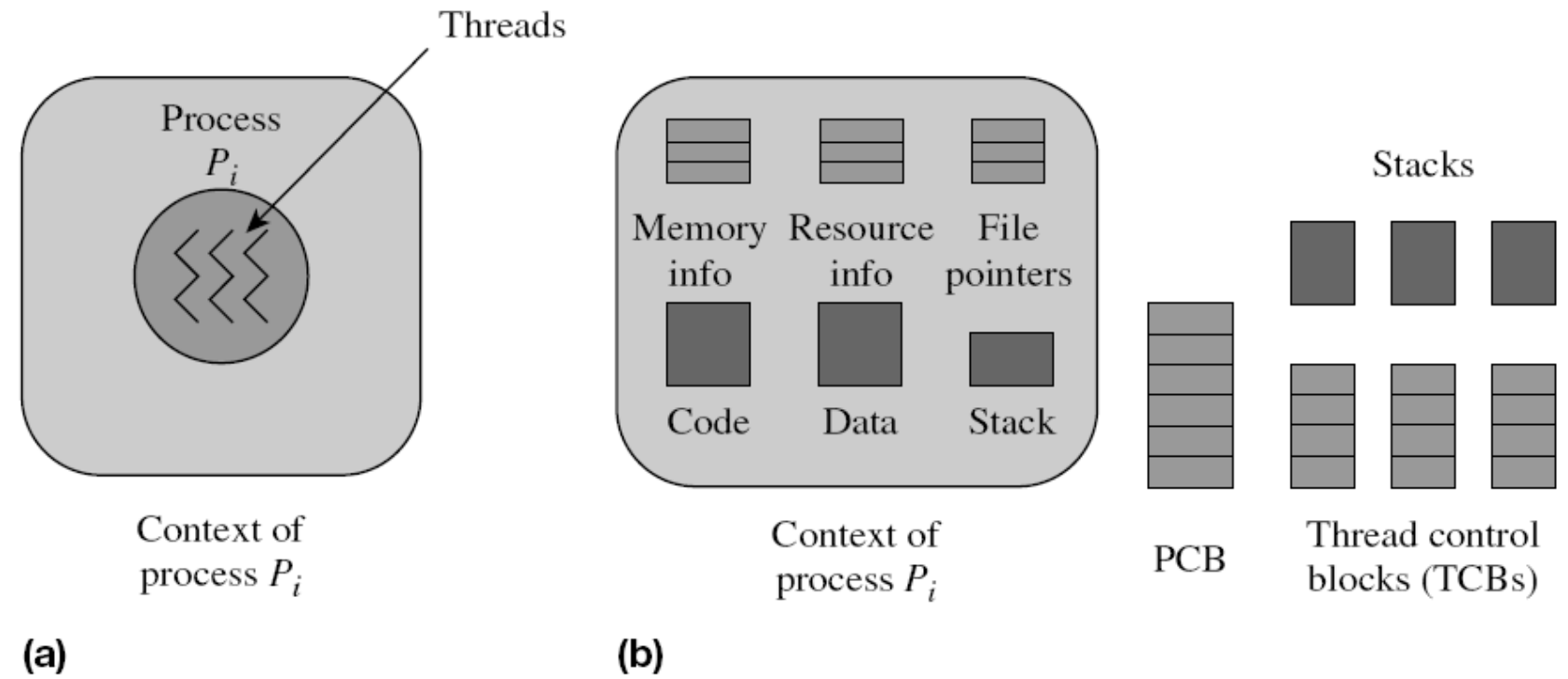

Figure 5.11 Threads in process P_i: (a) concept; (b) implementation.

Table 5.8 Advantages of Threads over Processes

Advantage	Explanation
Lower overhead of creation and switching	Thread state consists only of the state of a computation. Resource allocation state and communication state are not a part of the thread state, so creation of threads and switching between them incurs a lower overhead.
More efficient communication	Threads of a process can communicate with one another through shared data, thus avoiding the overhead of system calls for communication.
Simplification of design	Use of threads can simplify design and coding of applications that service requests concurrently.

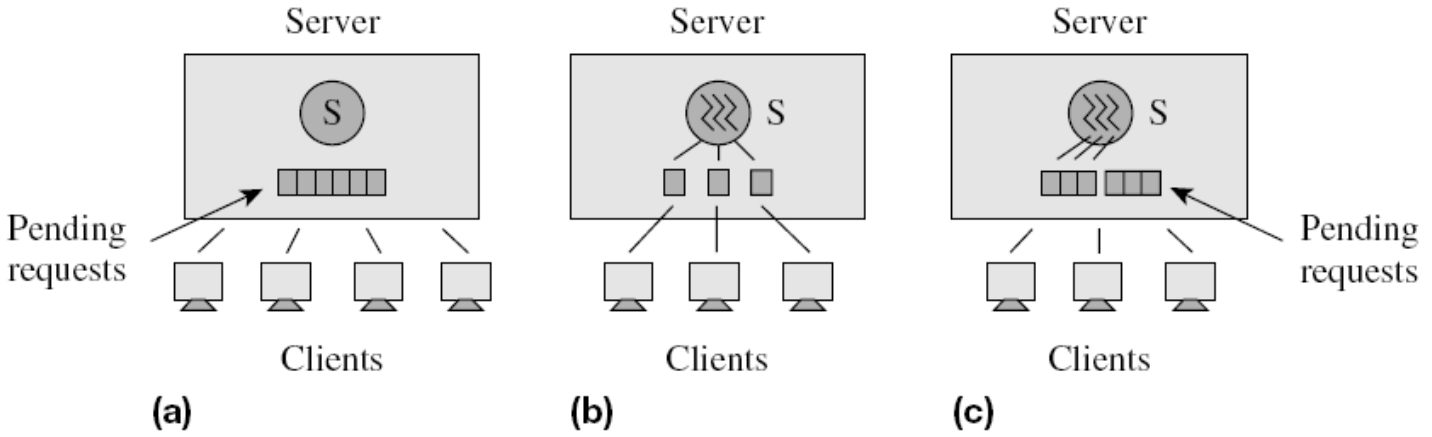

Figure 5.12 Use of threads in structuring a server: (a) server using sequential code; (b) multithreaded server; (c) server using a thread pool.

Coding for use of threads

- Use thread safe libraries to ensure correctness of data sharing
- Signal handling: which thread should handle a signal?
  - Choice can be made by kernel or by application
    - A synchronous signal should be handled by the thread itself
    - An asynchronous signal can be handled by any thread of the process
      - Ideally highest priority thread should handle it

POSIX Threads

- The ANSI/IEEE Portable Operating System Interface (POSIX) standard defines pthreads API
  - For use by C language programs
  - Provides 60 routines that perform the following:
    - Thread management
    - Assistance for data sharing–mutual exclusion
    - Assistance for synchronization–condition variables
  - A pthread is created through the call

    `pthread_create`*(< data structure >,< attributes >, < start routine >,< arguments >)*
  - Parent-child synchronization is through pthread_join
  - A thread terminates pthread_exit call

```
#include <pthread.h>
#include <stdio.h>
int size, buffer[100], no_of_samples_in_buffer;
int main()
{
   pthread_t id1, id2, id3;
   pthread_mutex_t buf_mutex, condition_mutex;
   pthread_cond_t buf_full, buf_empty;
   pthread_create(&id1, NULL, move_to_buffer, NULL);
   pthread_create(&id2, NULL, write_into_file, NULL);
   pthread_create(&id3, NULL, analysis, NULL);
   pthread_join(id1, NULL);
   pthread_join(id2, NULL);
   pthread_join(id3, NULL);
   pthread_exit(0);
}
void *move_to_buffer()
{
   /* Repeat until all samples are received */
   /* If no space in buffer, wait on buf_full */
   /* Use buf_mutex to access the buffer, increment no. of samples */
   /* Signal buf_empty */
   pthread_exit(0);
}
void *write_into_file()
{
   /* Repeat until all samples are written into the file */
   /* If no data in buffer, wait on buf_empty */
   /* Use buf_mutex to access the buffer, decrement no. of samples */
   /* Signal buf_full */
   pthread_exit(0);
}
void *analysis()
{
   /* Repeat until all samples are analyzed */
   /* Read a sample from the buffer and analyze it */
   pthread_exit(0);
}
```

Figure 5.13 Outline of the data logging application using POSIX threads.

Kernel-Level, User-Level, and Hybrid Threads

- Kernel-Level Threads
  - Threads are managed by the kernel
- User-Level Threads
  - Threads are managed by thread library
- Hybrid Threads
  - Combination of kernel-level and user-level threads

Kernel-Level Threads

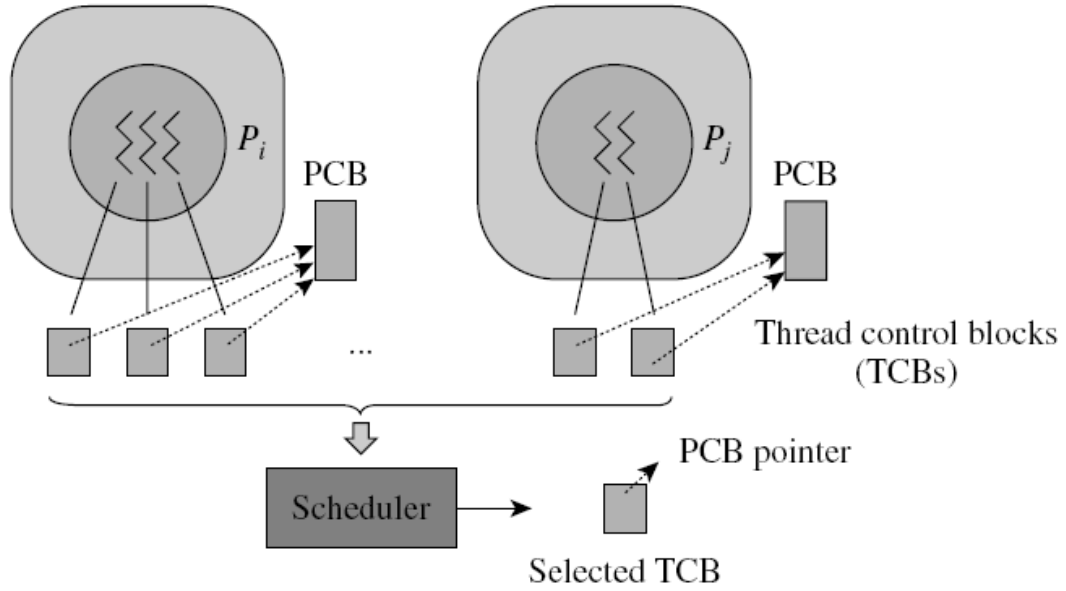

Figure 5.14 Scheduling of kernel-level threads.

- A kernel-level thread is like a process except that it has a smaller amount of state information
- Switching between threads of same process incurs the overhead of event handling

User-Level Threads

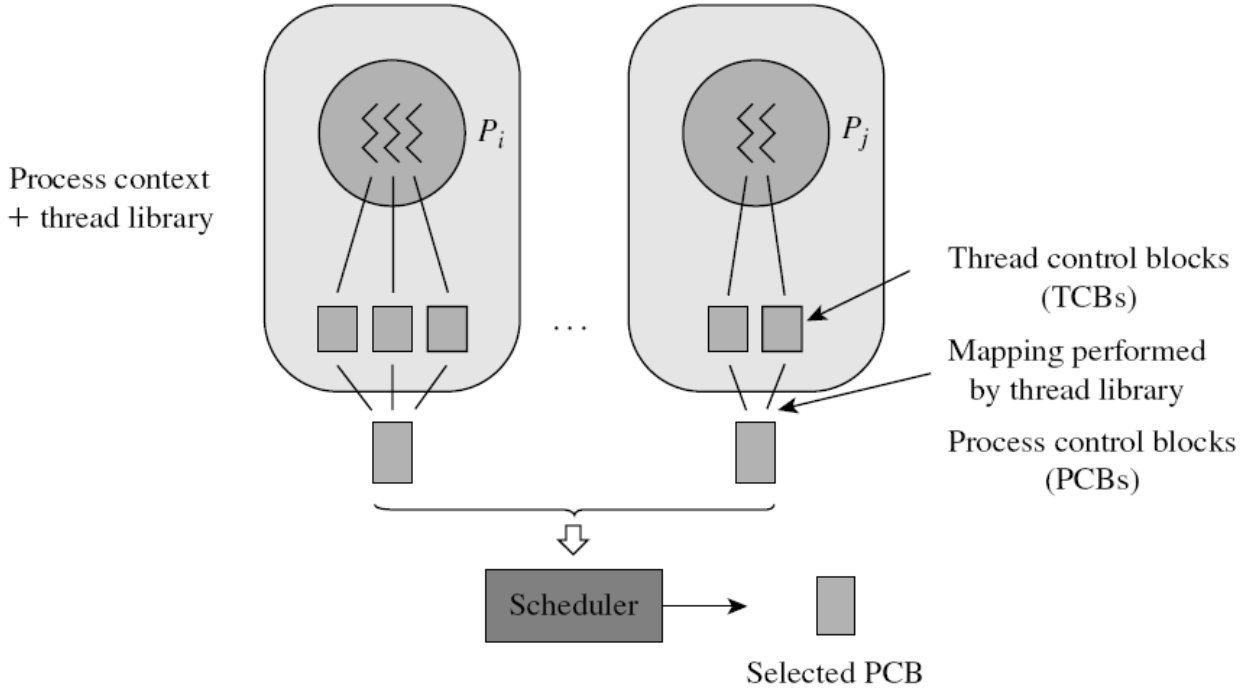

Figure 5.15 Scheduling of user-level threads.

- Fast thread switching because kernel is not involved
- Blocking of a thread blocks all threads of the process
- Threads of a process: No concurrency or parallelism

Scheduling of User-Level Threads

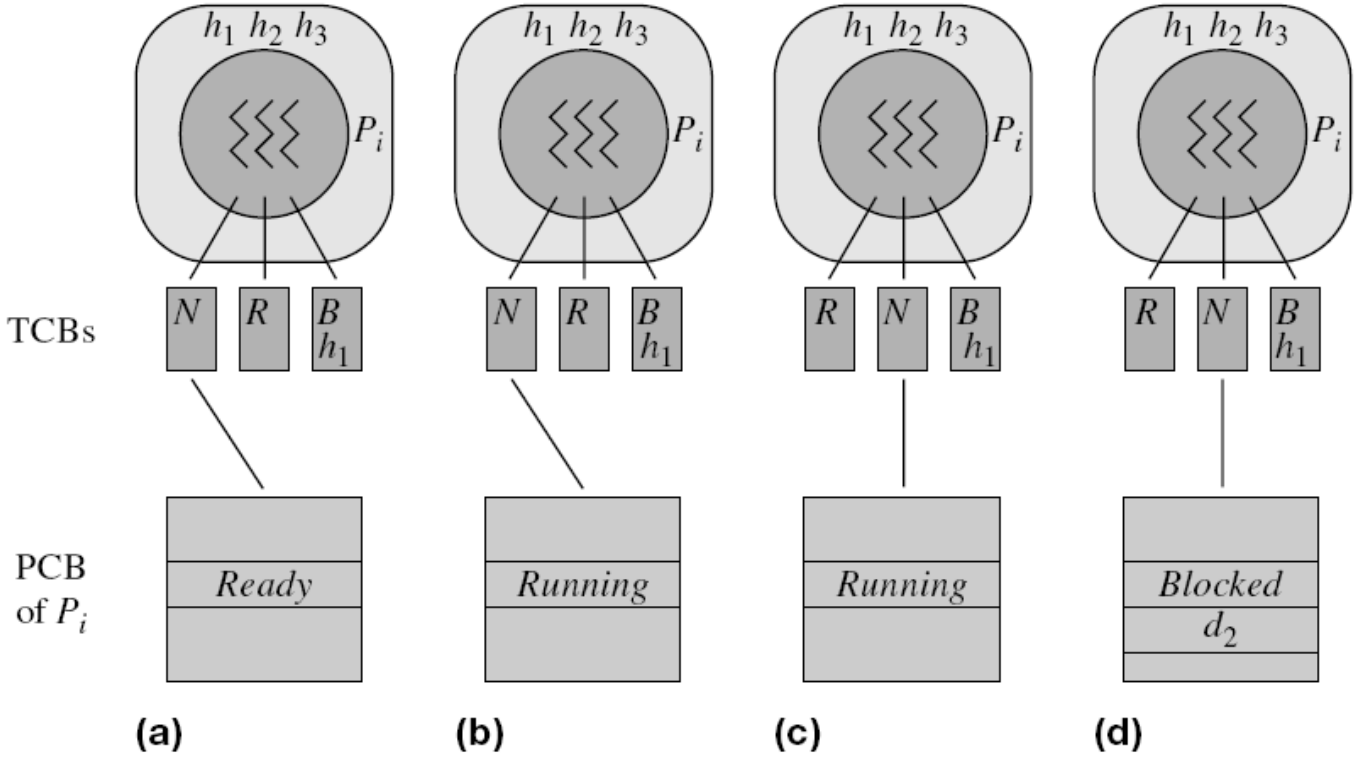

Figure 5.16 Actions of the thread library (N, R, B indicate *running*, *ready*, and *blocked*).

- Thread library maintains thread state, performs switching

Hybrid Thread Models

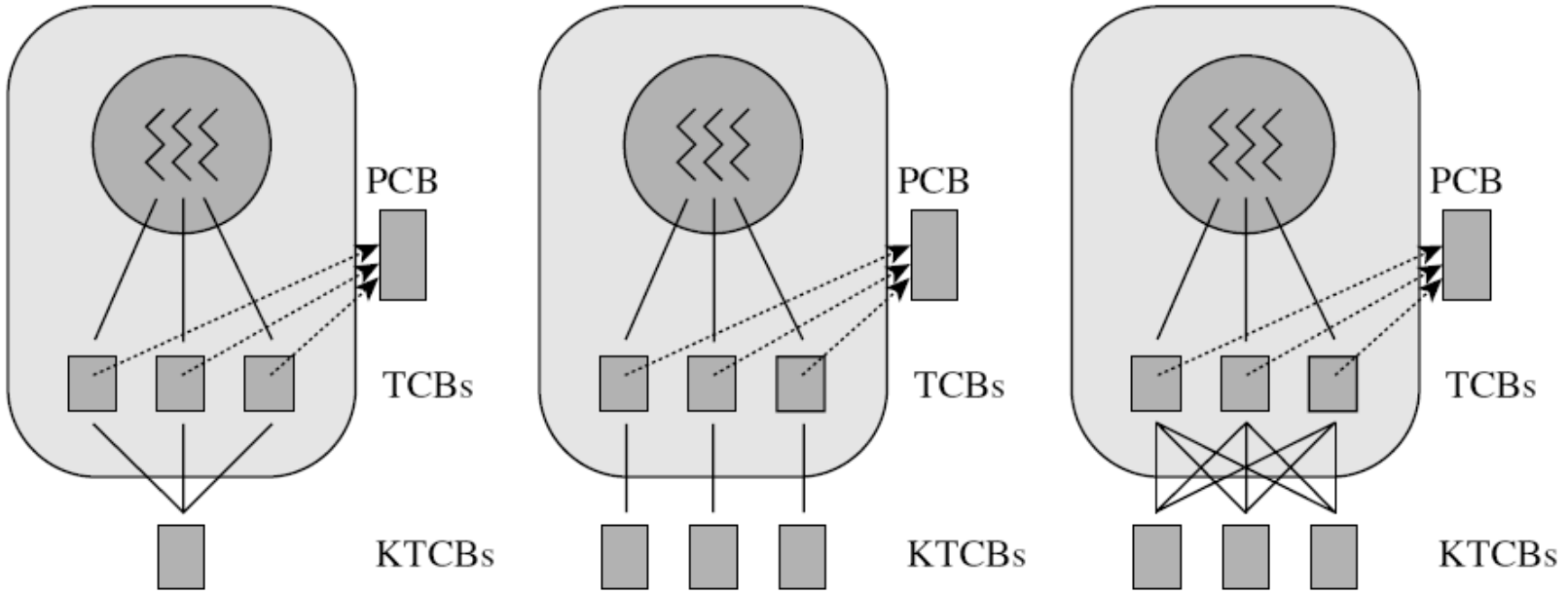

Figure 5.17 (a) Many-to-one; (b) one-to-one; (c) many-to-many associations in hybrid threads.

- Can provide a combination of parallelism and low overhead

Case Studies of Processes and Threads

- Processes in Unix
- Processes and Threads in Linux
- Threads in Solaris
- Processes and Threads in Windows

Processes in Unix

- Process executes kernel code on an interrupt or system call, hence kernel running and user running states
- A process P_i can wait for the termination of a child process through the system call *wait*

```
main()
{
        int saved_status;
        for (i=0; i<3; i++)
        {
             if (fork()==0)
             { /* code for child processes */
                  ...
                  exit();
             }
        }
        while (wait(&saved_status) !=-1);
               /* loop till all child processes terminate */
}
```

Figure 5.18 Process creation and termination in Unix.

Processes in Unix (continued)

Table 5.9 Interesting Signals in Unix

Signal	Description
SIGCHLD	Child process died or suspended
SIGFPE	Arithmetic fault
SIGILL	Illegal instruction
SIGINT	Tty interrupt (Control-C)
SIGKILL	Kill process
SIGSEGV	Segmentation fault
SIGSYS	Invalid system call
SIGXCPU	CPU time limit is exceeded
SIGXFSZ	File size limit is exceeded

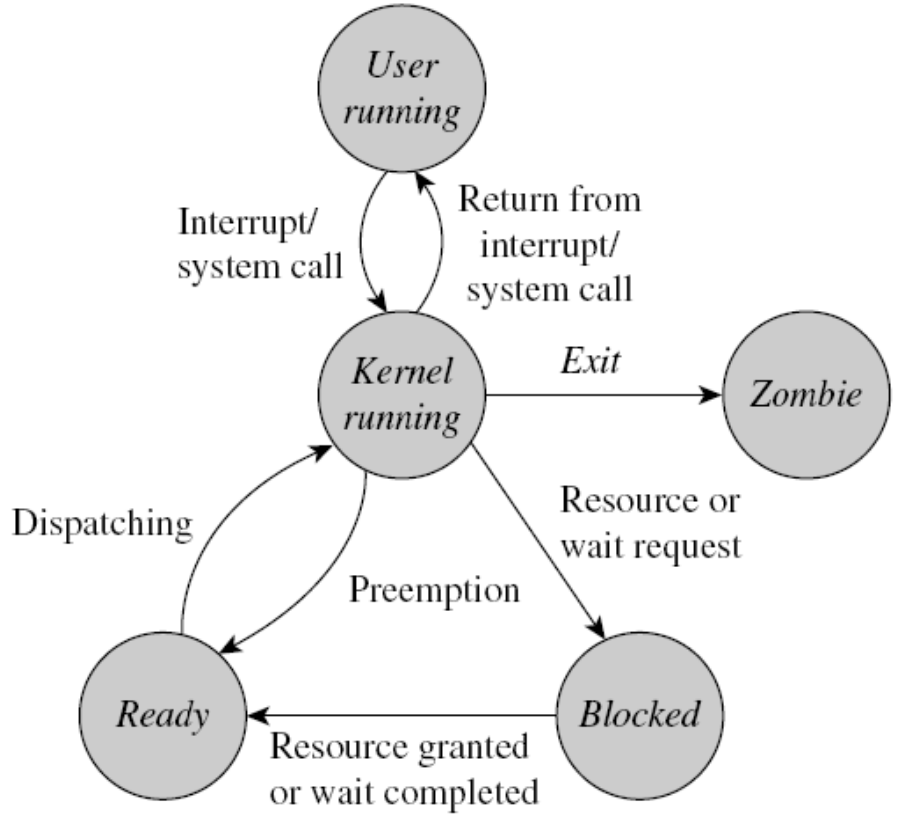

Figure 5.19 Process state transitions in Unix.

Processes and Threads in Linux

- Process states: Task_running, Task_interruptible, Task-uninterruptible, task_stopped and task_zombie
- Information about parent and child processes or threads is stored in a **`task struct`**

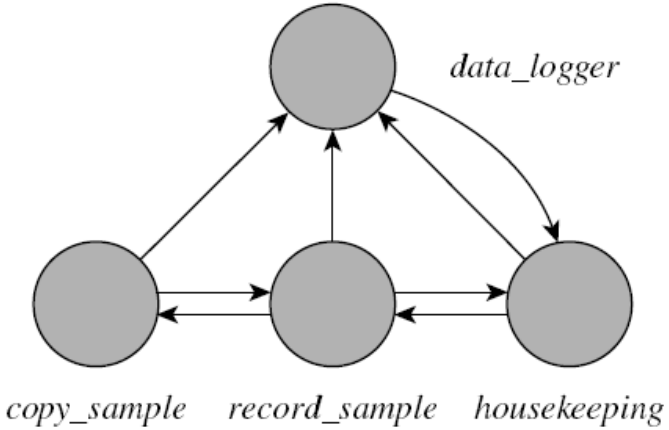

Figure 5.20 Linux process tree for the processes of Figure 5.2(a).

- Linux 2.6 supports kernel-level threads

Threads in Solaris

- Three kinds of entities govern concurrency and parallelism within a process:
  - User threads
  - Lightweight processes (LWPs)
    - Provides arallelism within a process
    - User thread are mapped into LWPs
  - Kernel threads
- Supported two different thread models
  - M x N model upto solaris 8
  - 1 : 1 model Solaris 8 onwards
- Provides scheduler activations to avoid thread blocking and notify events

Threads in Solaris (continued)

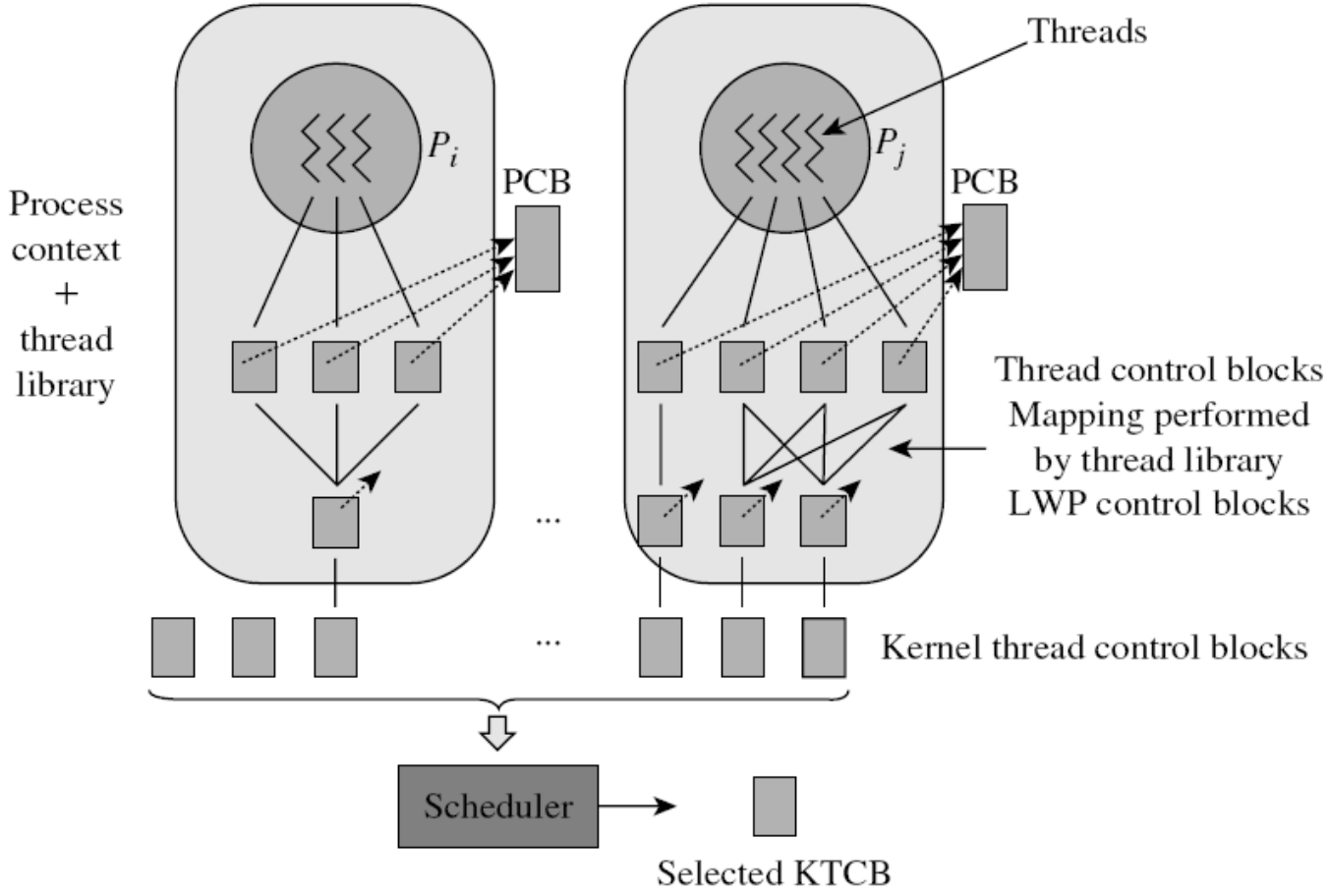

Figure 5.21 Threads in Solaris.

Processes and Threads in Windows

- Each process has at least one thread in it.
- Uses three control blocks per process
  - Executive process block: process id, a kernel process block and address of process environment block
  - Kernel process block: process state, KTB addresses
  - Process environment block: information about code and heap
- Uses three thread blocks per thread
  - Executive thread block contains pointer to kernel thread block and executive process block
  - Kernel thread block: stack, state and kernel environment block

Processes and Threads in Windows (continued)

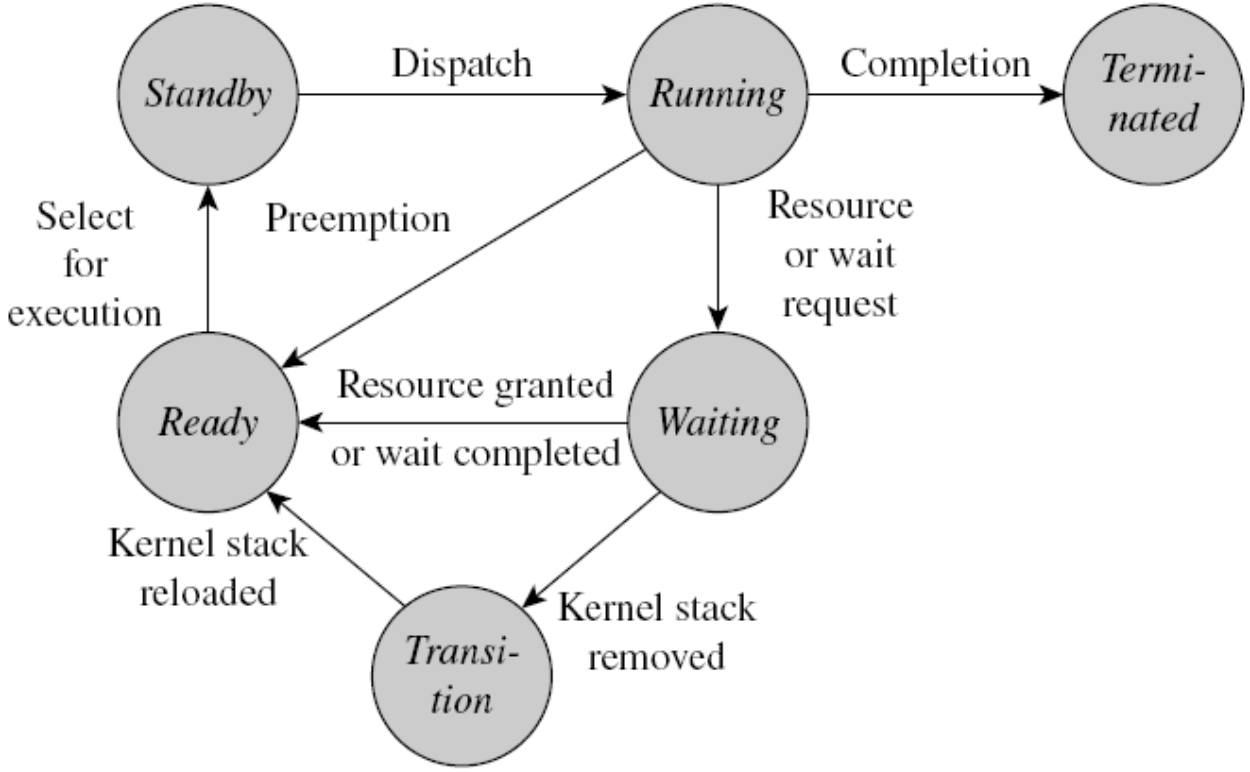

Figure 5.22 Thread state transitions in Windows.

Summary

- Execution of a program can be speeded up through either *parallelism* or *concurrency*
- A *process* is a model of execution of a program
  - Can create other processes by making requests to the OS through system calls
    - Each of these processes is called its *child process*
    - Provides parallelism or concurrency
- OS provides *process synchronization* means
- OS allocates resources to a process and stores information about them in the *process context* of the process

Summary (continued)

- To control operation of the process, OS uses notion of a *process state*
  - *Ready, running, blocked, terminated*, *suspended*
- OS keeps information concerning each process in a *process control block* (PCB)
  - Process state and CPU state associated with process
  - Scheduler selects a *ready* process and dispatcher switches CPU to selected process through information found in its process context and the PCB

Summary (continued)

- A *thread* is an alternative model of execution of a program
  - Overhead of switching between threads is much less than the overhead of switching between processes
- Three models of threads:
  - *Kernel-level threads*
  - *User-level threads*
  - *Hybrid threads*
- Thread models have different implications for switching overhead, concurrency, and parallelism