Lecture 5

Threads & CPU scheduling I

Lecture Information

Ceng328 *Operating Systems* at March 16, 2010

Dr. Cem Özdoğan
Computer Engineering Department
Çankaya University

Contents

Threads
Overview
Motivation
Benefits
Multithreading Models
Many-to-One Model
One-to-One Model
Many-to-Many Model
Thread Libraries
Pthreads
Threading Issues
The *fork*() and *exec*() System Calls
Cancellation
Signal Handling
Operating-System Examples
Linux Threads
CPU scheduling
Basic Concepts
CPU-I/O Burst Cycle
CPU Scheduler
Pre-emptive Scheduling
Dispatcher
Scheduling Criteria

Overview I

- A traditional (or heavyweight) process has a single thread of control.
- A thread (also referred to as a *light-weight process LWP*) is a basic unit of CPU utilization.
- All threads in a process have exactly the same address space, which means that they also share the same global variables.
- It shares with other threads belonging to the same process its code section, data section, and other OS resources, such as open files and signals (see Fig. 1).

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure: The first column lists some items shared by all threads in a process (process properties). The second one lists some items private to each thread.

Overview II

- Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.
- If a process has multiple threads of control in the same address space running in quasi-parallel, as though they were separate processes (except for the shared address space).

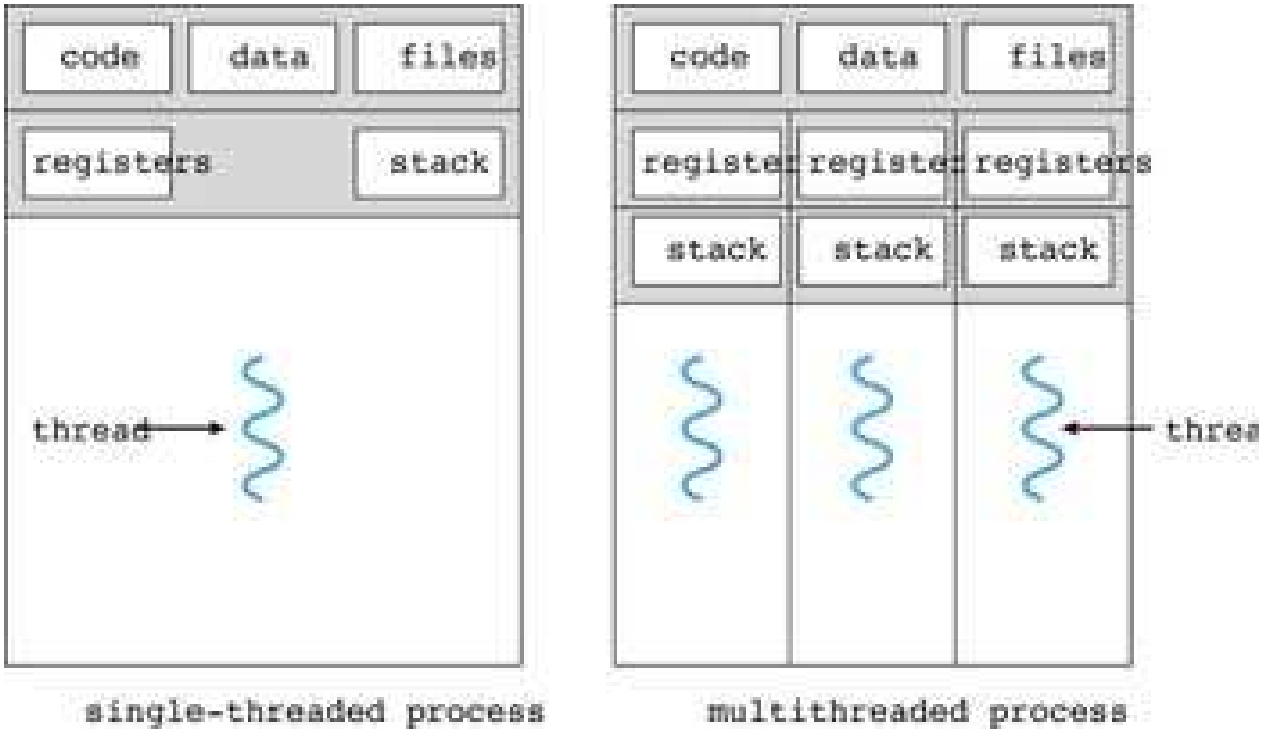


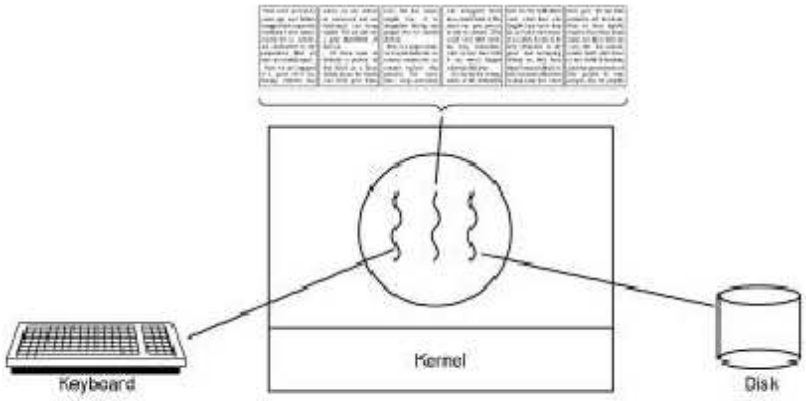
Figure: Single-threaded and multithreaded processes.

Overview III

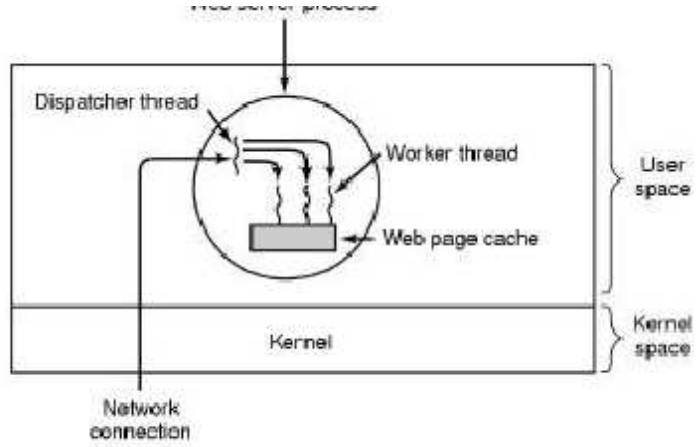
- Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately.
  - The threads share an address space, open files, and other resources.
  - The processes share physical memory, disks, printers, and other resources.
- Since every thread can access every memory address within the process' address space, there is no protection between threads because
  - it is impossible,
  - it should not be necessary. They are *cooperating*, *not competing*.
- Like a traditional process (i.e., a process with only one thread), a thread can be in any one of several states.
- The transitions between thread states are the same as the transitions between process states.

Motivation

- An application typically is implemented as a separate process with several threads of control.
  - A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.
  - A web browser might have one thread display images or text while another thread retrieves data from the network.



A word processor with three threads



A multithreaded Web server

Figure: Left: A word processor with three threads. Right: A multithreaded web server.

Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
2. **Resource sharing**. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy of Overheads**. Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.
4. **Utilization of multiprocessor architectures**. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors (real parallelism).

Multithreading Models I

- Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.
  - User threads are supported above the kernel and are managed without kernel support,
  - whereas kernel threads are supported and managed directly by the OS.

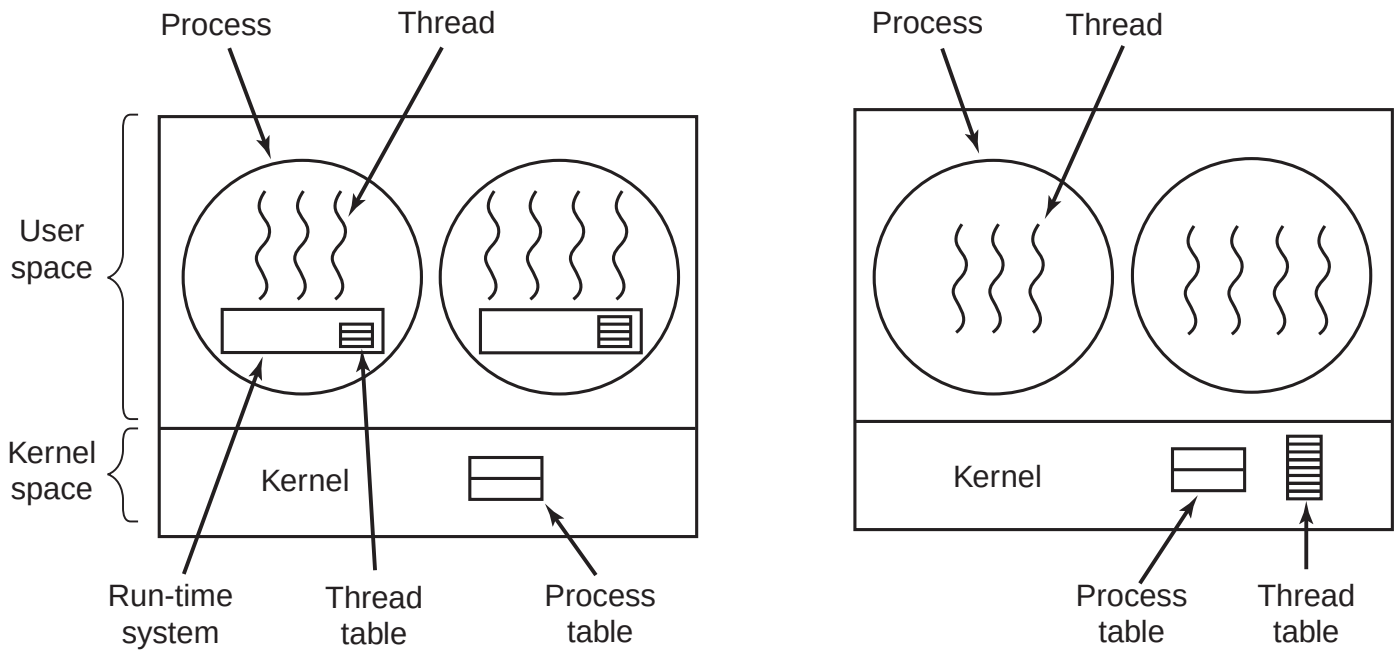


Figure: (a) A user-level threads package. (b) A threads package managed by the kernel.

Multithreading Models II

Implementing Threads in User Space:

- The threads package entirely in user space (see Fig. 4a). The kernel knows nothing about them.
- The first, and most obvious, advantage is that a user-level threads package can be implemented on an OS that does not support threads.
- Among other issues, no trap is needed, no context switch is needed, the memory cache need not be flushed, and so on. This makes thread scheduling very fast.
- Despite their better performance, user-level threads packages have a major problem as if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU.
- While user threads usually have lower management load compared to kernel threads, one must consider this in relation to their lower functionality.

Multithreading Models III

Implementing Threads in the Kernel:

- Supported by the kernel, the kernel performs all management (creation, scheduling, deletion, etc., see Fig. 4b).
- There is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system.
- if one thread blocks, another may be run. In addition, if one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads.
- Ultimately, there must exist a relationship between user threads and kernel threads.

Many-to-One Model

- The many-to-one model (see Fig. 5) maps many user-level threads to one kernel thread.

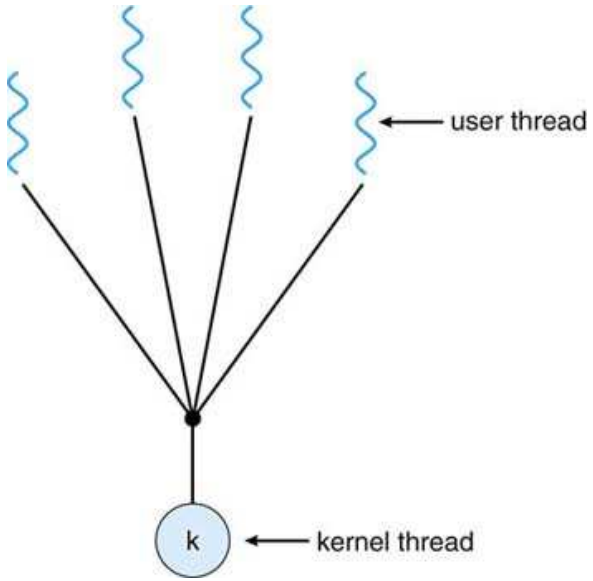


Figure: Many-to-one model.

- Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

One-to-One Model

- The one-to-one model (see Fig. 6) maps each user thread to a kernel thread.

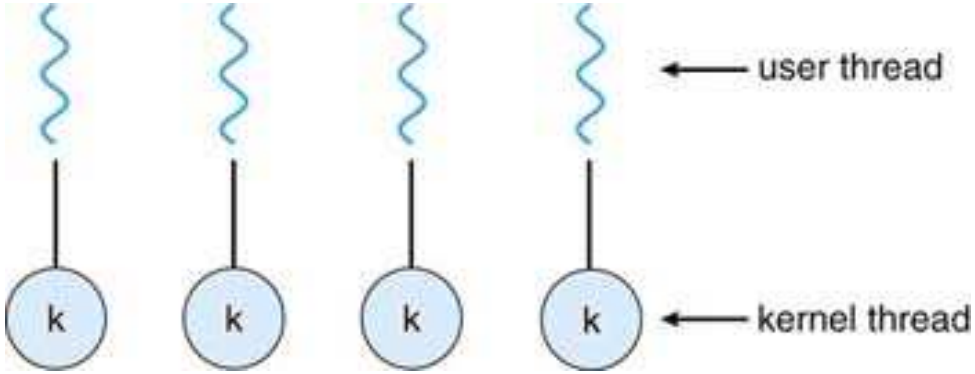


Figure: One-to-one model.

- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- Overhead of creating kernel threads can degrade the performance of an application.
- Linux, along with the family of Windows OSs implement the one-to-one model.

Many-to-Many Model

- The many-to-many model (see Fig. 7) multiplexes many user-level threads to a smaller or equal number of kernel threads.

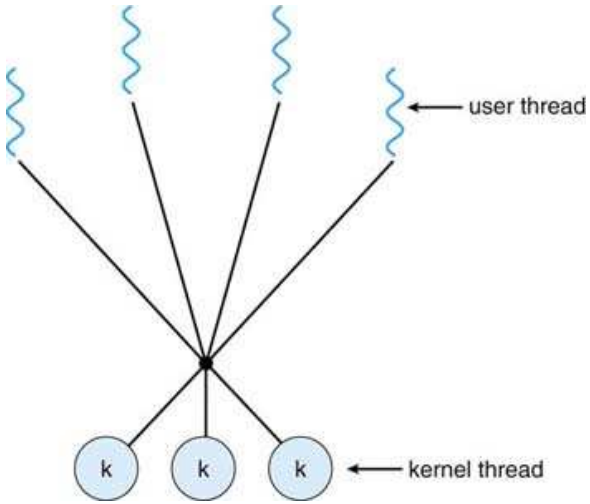


Figure: Many-to-many model.

- The many-to-many model suffers from neither of these shortcomings:
  - Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
  - Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.
- Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.

Thread Libraries

- Three main thread libraries are in use today:
  1. **POSIX Pthreads**. Pthreads, the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library.
  2. **Win32**. The Win32 thread library is a kernel-level library available on Windows systems.
  3. **Java**. The Java thread API allows thread creation and management directly in Java programs.
     - However, because in most instances the JVM is running on top of a host OS, the Java thread API is typically implemented using a thread library available on the host system.

Pthreads I

- Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.
- This is a specification for thread behavior, not an implementation.

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure: Some of the Pthreads function calls.

- A common thread call is *thread_yield*, which allows a thread to voluntarily give up the CPU to let another thread run.
  - Such a call is important because there is *no clock interrupt to actually enforce time-sharing as there is with processes*.
  - Thus it is important for threads to be polite and voluntarily surrender the CPU from time to time to give other threads a chance to run.

Pthreads II

```
#include <pthread.h>
#include <stdio.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */
int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */
  if (argc != 2) {
    fprintf(stderr, "usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv [1]) < 0) {
    fprintf(stderr,"%d must be >= O\n",atoi(argv[1]));
    return -1;
  }
  /* get the default attributes */
  pthread_attr_init (&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]) ;
  /* wait for the thread to exit */
  pthread_join(tid,NULL) ;
  printf (" sum = %d\n", sum) ;
}
```


Pthreads III

- The C program shown above and below demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a nonnegative integer in a separate thread (do not forget to compile with *-lpthread* flag.).
- In a Pthreads program, separate threads begin execution in a specified function (in this program; *runner*()).

```
/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;
  for (i = 1; i <= upper; i++)
    sum += i;
  pthread_exit(0) ;
}
```


The *fork*() and *exec*() System Calls

- If one thread in a program calls *fork*(),
  - does the new process duplicate all threads,
  - or is the new process single-threaded?
- Some UNIX systems have chosen to have two versions of *fork*(),
  - one that duplicates all threads
  - and another that duplicates only the thread that invoked the *fork*() system call.
- Which of the two versions of *fork*() to use depends on the application.
  - If *exec*() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to *exec*() will replace the process. In this instance, duplicating only the calling thread is appropriate.
  - If, however, the separate process does not call *exec*() after forking, the separate process should duplicate all threads.

Cancellation I

- **Thread cancellation** is the task of terminating a thread before it has completed.
  - For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
  - Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further.
- A thread that is to be canceled is often referred to as the **target thread**.

Cancellation II

- Cancellation of a target thread may occur in two different scenarios:

1. **Asynchronous cancellation**. One thread immediately terminates the target thread.
   - The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread
   - or where a thread is canceled while in the midst of updating data it is sharing with other threads.
   - Often, the OS will reclaim system resources from a canceled thread but will not reclaim all resources.
2. **Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.
   - With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine if it should be canceled or not.
   - This allows a thread to check whether it should be canceled at a point when it can be canceled safely.
   - Pthreads refers to such points as cancellation points.

Signal Handling I

- A signal is used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received either **synchronously** or **asynchronously**, depending on the source of and the reason for the event being signaled.
  1. A signal is generated by the occurrence of a particular event.
  2. A generated signal is delivered to a process.
  3. Once delivered, the signal must be handled.
- Examples of **synchronous** signals include illegal memory access and division by O.
- Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).

Signal Handling II

- When a signal is generated by an event external to a running process, that process receives the signal **asynchronously**.
- Examples of such signals include terminating a process with specific keystrokes (such as $< control >< C >$ and having a timer expire.
- Typically, an asynchronous signal is sent to another process.
- Every signal may be handled by one of two possible handlers:
  - A default signal handler.
  - A user-defined signal handler
- Every signal has a **default signal handler** that is run by the kernel when handling that signal.
- This default action can be overridden by a **user-defined signal handler** that is called to handle the signal.

Linux Threads I

- Linux provides the ability to create threads using the *clone*() system call (*fork*() system call for duplicating a process) .
- In fact, Linux generally uses the term **task** -rather than process or thread - when referring to a flow of control within a program.
- When *clone*() is invoked, it is passed a set of flag. Some of these flags are listed in Fig. 9 below:

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Figure: Some flags for *clone*() system call.

Linux Threads II

- if *clone*() is passed the flags above in the Fig. 9, the parent and child tasks will share the same mentioned resources.
- Using *clone*() in this fashion is equivalent to *creating a thread*.
- However, if none of these flags are set when *clone*() is invoked, no sharing takes place, resulting in functionality similar to that provided by the *fork*() system call.

Basic Concepts

- In multiprogramming systems, whenever two or more processes are simultaneously in the *ready state*, a choice has to be made which process to run next.
  - The part of the OS that makes the choice is called the **scheduler**
  - and the algorithm it uses is called the **scheduling algorithm**.
- Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources.
- Thus, its scheduling is central to OS design.
- Many of the same issues that apply to process scheduling also apply to thread scheduling, although some are different.

CPU-I/O Burst Cycle I

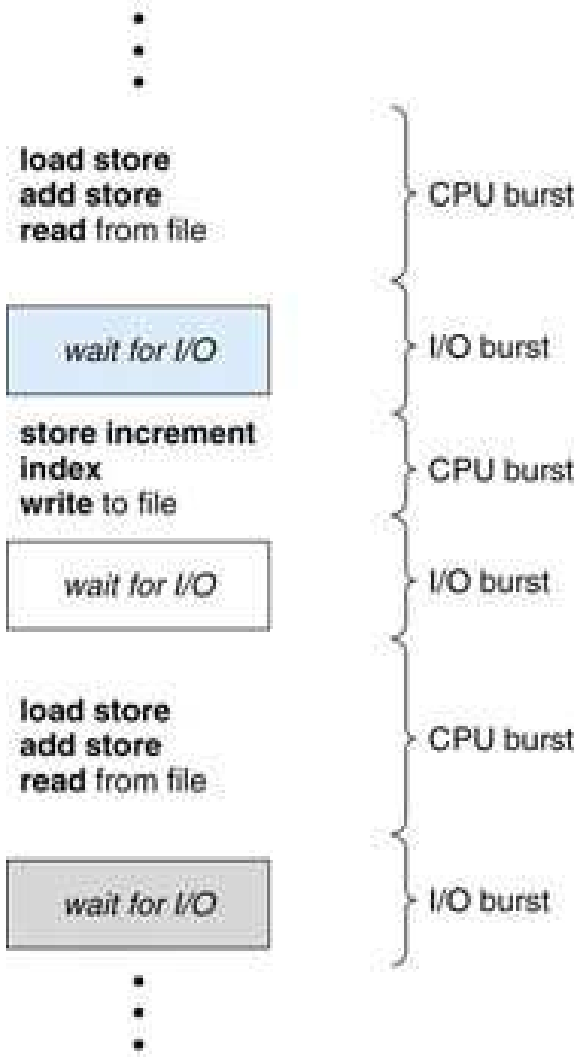
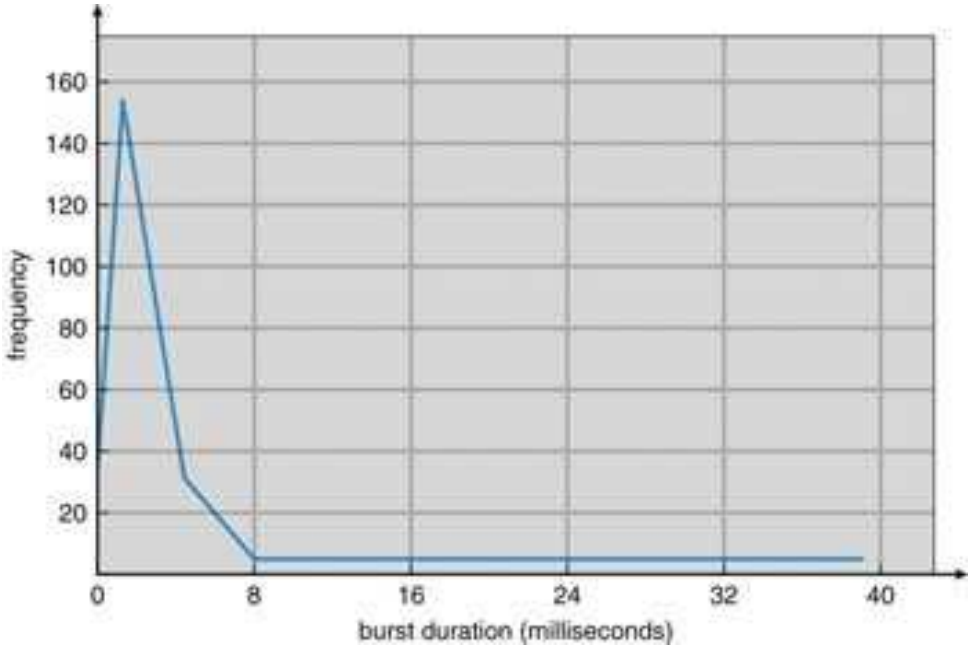


Figure: Alternating sequence of CPU and I/O bursts.

- The success of CPU scheduling depends on an observed property of processes:
  - Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states.
  - Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution (see Fig. 10).

CPU-I/O Burst Cycle II

- The durations of CPU bursts have a frequency curve similar to that shown in Fig. 11.



- An **I/O-bound** program typically has many short CPU bursts.
- A **CPU-bound** program might have a few long CPU bursts.

Figure: Histogram of CPU-burst durations.

- The curve is generally characterized as exponential, with a large number of short CPU bursts and a small number of long CPU bursts.
- This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

CPU-I/O Burst Cycle III

- Nearly all processes alternate bursts of computing with (disk) I/O requests, as shown in Fig. 12.

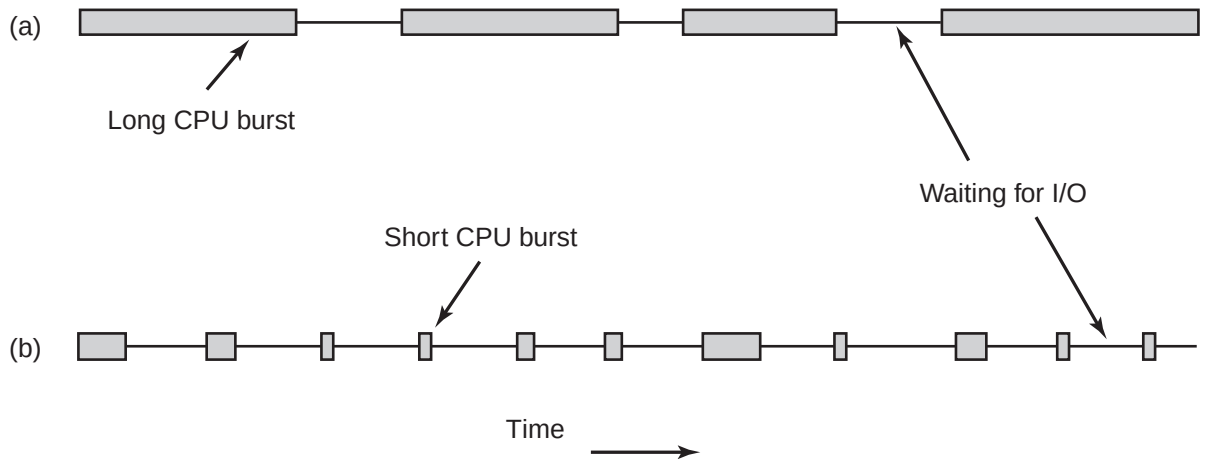


Figure: Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

- Having some CPU-bound processes and some I/O-bound processes in memory together is a better idea than first loading and running all the CPU-bound jobs and then when they are finished loading and running all the I/O-bound jobs (**a careful mix of processes**).

CPU Scheduler

- Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the short-term scheduler (or CPU scheduler).
- The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.
- Conceptually all the processes in the ready queue are lined up waiting for a chance to run on the CPU.
- The records in the queues are generally process control blocks (PCBs) of the processes.

Pre-emptive Scheduling I

- CPU-scheduling decisions may take place under the following four circumstances:
  1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes).
  2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
  3. When a process switches from the waiting state to the ready state (for example, at completion of I/O, on a semaphore, or for some other reason).
  4. When a process terminates. If no process is ready, a system-supplied idle process is normally run.
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- There is a choice, however, for situations 2 and 3.

Pre-emptive Scheduling II

- When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or cooperative;
- otherwise, it is **pre-emptive**.
- Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU voluntarily.
- Unfortunately, pre-emptive scheduling incurs a cost associated with access to shared data.
  - Consider the case of two processes that share data.
  - While one is updating the data, it is preempted so that the second process can run.
  - The second process then tries to read the data, which are in an inconsistent state.
  - In such situations, we need new mechanisms to coordinate access to shared data.

Pre-emptive Scheduling III

- A **nonpreemptive scheduling algorithm** picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU. First-Come-First-Served (FCFS), Shortest Job first (SJF).
- In contrast, a **pre-emptive scheduling algorithm** picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run. Round-Robin (RR), Priority Scheduling.

Pre-emptive Scheduling III

- Another component involved in the CPU-scheduling function is the **dispatcher**.
- The scheduler is concerned with deciding *policy*, not providing a *mechanism*.
- The dispatcher is the low-level mechanism (Responsibility: Context-switch).
  - Switching context,
  - Switching to user mode,
  - Jumping to the proper location in the user program to restart that program.
- The dispatcher should be as fast as possible, since it is invoked during every process switch.
- The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Scheduling Criteria I

- Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another.
- Which algorithm to use?, consider the properties;
  - **CPU utilization**. We want to keep the CPU as busy as possible.
  - **Throughput**. One measure of work is the number of processes that are completed per time unit, called throughput.
  - **Turnaround time**. The interval from the time of submission of a process to the time of completion is the turnaround time.
    - $T_r = T_s + T_w$, where T_s :*Execution time* and T_w :*Waiting time*.
  - **Waiting time**. The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue.
  - **Response time**. In an interactive system, turnaround time may not be the best criterion. Thus, another measure is the time from the submission of a request until the first response is produced.

Scheduling Criteria II

- One problem in selecting a set of performance criteria is that they often conflict with each other.
- For example, increased processor utilization is usually achieved by increasing the number of active processes, but then response time decreases.
- A scheduling algorithm that maximizes *throughput* may not necessarily minimize *turnaround* time.
  - Given a mix of short jobs and long jobs, a scheduler that always ran short jobs and never ran long jobs might achieve an excellent throughput (many short jobs per hour) but at the expense of a terrible turnaround time for the long jobs.
  - If short jobs kept arriving at a steady rate, the long jobs might never run, making the mean turnaround time infinite while achieving a high throughput.
- It is desirable to maximize **CPU utilization** and **throughput** and to minimize **turnaround time**, **waiting time**, and **response time**.

Scheduling Criteria III

- Some goals of the scheduling algorithm under different circumstances, see Fig. 13.

All systems
- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems
- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems
- Response time - respond to requests quickly
- Proportionality - meet users' expectations

Real-time systems
- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

Figure: Some goals of the scheduling algorithm under different circumstances.

- Under all circumstances, fairness is important.
- Another general goal is keeping all parts of the system busy when possible.