Operating Systems — Threads and Scheduling

ECE_3TC31_TP/INF107

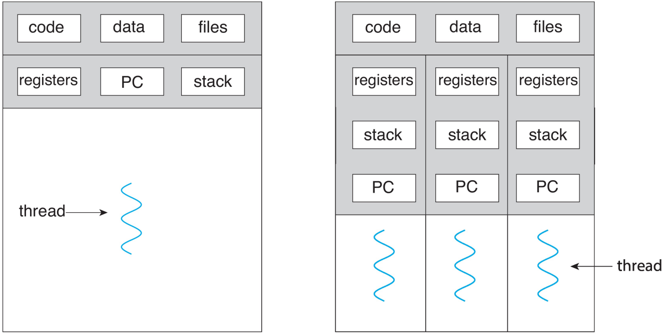
Stefano Zacchiroli

2024

Threads

Multithreading

- So far, we assumed each process had a single **thread of execution** ("thread" for short)
- Consider now having **multiple program counters** per process → **multithreading**
- OS must keep track of **thread-specific data**, including registers and stack



single-threaded process multithreaded process

- Note how, differently from processes, threads *share a single address space* → **memory is shared by default** among all threads of the same process

Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces in interactive applications
- **Resource Sharing** – threads share resources of process, easier than on-demand shared memory or message passing
- **Economy** – cheaper than process creation, thread switching has lower overhead than context switching
- **Scalability** – process can take advantage of multicore architectures

There are also **drawbacks**!

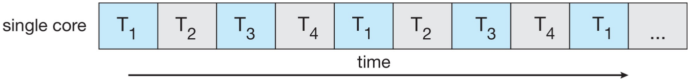
In particular it can be difficult to write *correct* multithreaded programs against the risk of race conditions.

We will explore this topic in the upcoming lecture about synchronization.

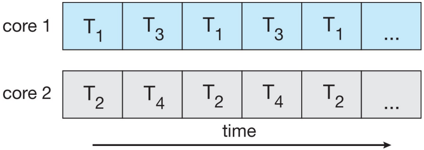
Multicore Programming

- On system with more than one core, multithreading may lead to multiple CPU instructions of the same program being executed *at the same time* → **parallelism**
- Beware of the difference between:
  - *Parallelism* implies a system can perform more than one task simultaneously
  - *Concurrency* supports more than one task making progress
    - OS can give the *illusion of parallelism* on a single processor/core, by alternating quickly between tasks

Concurrent execution on single-core system:



Parallelism on a multi-core system:



Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- Specification, not implementation
- API specifies behavior of the library, implementation is up to development of the library
- Common in UNIX operating systems

You will learn more about pthreads in the upcoming lab session; in the following we will just briefly walk through a phtread hello-world-style example.

Pthreads — Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 5

void *perform_work(void *arguments){
    int index = *((int *)arguments);
    int sleep_time = 1 + rand() % NUM_THREADS;
    printf("THREAD %d: Started.\n", index);
    printf("THREAD %d: Will be sleeping for %d seconds.\n", index, sleep_time);
    sleep(sleep_time);
    printf("THREAD %d: Ended.\n", index);
    return NULL;
}
```

(source)

Pthreads — Example (cont.)

```c
int main(void) {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    int i;
    int result_code;

    for (i = 0; i < NUM_THREADS; i++) {  // Create all threads one by one
        printf("IN MAIN: Creating thread %d.\n", i);
        thread_args[i] = i;
        result_code = pthread_create(&threads[i], NULL, perform_work, &thread_args[i]);
        assert(!result_code);
    }
    printf("IN MAIN: All threads are created.\n");

    for (i = 0; i < NUM_THREADS; i++) {  // Wait for each thread to complete
        result_code = pthread_join(threads[i], NULL);
        assert(!result_code);
        printf("IN MAIN: Thread %d has ended.\n", i);
    }
    printf("MAIN program has ended.\n");
    return 0;
}
```

Pthreads — Example (cont.)

```
$ gcc -Wall pthreads-hello.c -o pthreads-hello -pthread

$ ./pthreads-hell
IN MAIN: Creating thread 0.
IN MAIN: Creating thread 1.
IN MAIN: Creating thread 2.
IN MAIN: Creating thread 3.
IN MAIN: Creating thread 4.
THREAD 0: Started.
THREAD 0: Will be sleeping for 4 seconds.
IN MAIN: All threads are created.
THREAD 1: Started.
THREAD 1: Will be sleeping for 2 seconds.
THREAD 2: Started.
THREAD 2: Will be sleeping for 1 seconds.
THREAD 4: Started.
THREAD 4: Will be sleeping for 3 seconds.
THREAD 3: Started.
THREAD 3: Will be sleeping for 4 seconds.
THREAD 2: Ended.
THREAD 1: Ended.
THREAD 4: Ended.
THREAD 0: Ended.
THREAD 3: Ended.
IN MAIN: Thread 0 has ended.
IN MAIN: Thread 1 has ended.
IN MAIN: Thread 2 has ended.
IN MAIN: Thread 3 has ended.
IN MAIN: Thread 4 has ended.
MAIN program has ended.
```

Are Threads and Processes that Different? — The Linux Example

- The Linux kernel refers to executable entities as “**tasks**” rather than threads or processes
- As we have seen last week, process creation is requested using the `fork()` system call
- Thread creation is requested through the `clone()` system call
- `clone()` flags allow a parent to **selectively share, or not, resources with its child**:

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.

- Intuition (“VM” stands for “virtual memory” here):
  - `CLONE_VM` present → “new thread”
  - `CLONE_VM` absent → “new process”
- `struct task_struct` (recursively) points to task data structures (shared or unique)

Bottom line: the distinction between threads and processes is not clear cut, but rather a matter of which resources executable entities decide to share.

Scheduling

Reminder

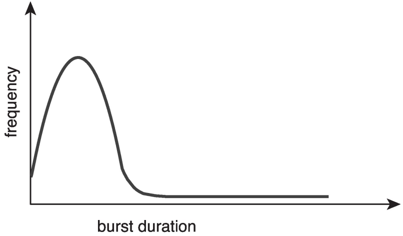
We have seen in a previous lecture that:

- With *multiprogramming*, several programs are loaded into memory at the same time
- *Processes* pass through several states (running, waiting, ready, etc.) during their lifetimes
- At any given time *a maximum of one process* (per CPU core) can be in execution
- **Scheduling** is the OS activity deciding *which process is in execution at a given time* (on each core)
- The **process scheduler** (or *CPU scheduler*) selects among available processes[1] for next execution on a given CPU core
  - *Ready queue:* set of all processes residing in main memory, ready and waiting to execute
  - *Wait queues:* set of processes waiting for an event
  - Processes migrate among the various queues as they change state

[1] Actually: “threads” or, more generally, “runnable entities”. We will use “process” for simplicity in the following slides, although what is actually scheduled are runnable entities.

CPU and I/O Bursts

- During execution a process alternates between **CPU bursts** and **I/O bursts**
  - Cycle of CPU execution and *waiting* for I/O
  - If CPU bursts dominate performances the process is said to be **CPU bound**, otherwise **I/O bound**
- The distribution of CPU burst duration is of main concern for scheduling decisions. Experimental results show that there are:
  - Large number of short CPU bursts
  - Small number of longer CPU bursts



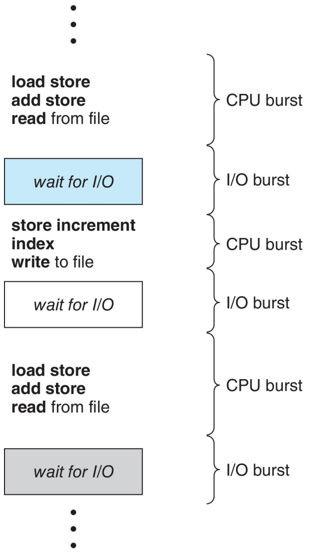
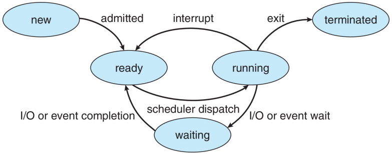


Figure: typical process lifetime

CPU Scheduler

- The **CPU scheduler** selects from among the processes in *ready queue*, and allocates a CPU core to one of them
  - Queue may be ordered in various ways ← important policy decision for the scheduler
- CPU scheduling decisions may take place at the following state transitions:
  1. running → waiting
  2. running → terminates
  3. running → ready
  4. waiting → ready



- For (1) and (2), a new process (if one exists in the ready queue) must be selected for execution.
- For (3) and (4), however, there is a choice.
  - If no change of scheduled process can happen → **nonpreemptive scheduling**
    - Once the CPU has been allocated to a process, the process keeps it until waiting or termination.
  - If a change of scheduled process can happen → **preemptive scheduling**
    - The OS can “take away” (= preempt) the CPU from one process and give it to another.

Scheduling Criteria and Goals

Several *metrics* are used as criteria to evaluate scheduling policies:

CPU utilization: keep the CPU as busy as possible

Throughput: number of processes that complete their execution per time unit

Turnaround time: amount of time to execute a particular process (until completion)

Waiting time: amount of time a process has been waiting in the ready queue

Response time: amount of time it takes from request submission until the *first* response is produced

Based on these metrics, general **optimization goals** for the scheduler are:

- Maximize CPU utilization
- Maximize throughput
- Minimize turnaround time
- Minimize waiting time
- Minimize response time

Several **scheduling policies** (or "algorithms") exist, with different trade-offs.

Let's look at the main ones.

First Come, First Served (FCFS) Scheduling

- Pure FIFO (First In, First Out) ordering of the ready queue
- Nonpreemptive

Process	Burst duration
P_1	24
P_2	3
P_3	3

- Suppose processes arrive in the order: P_1, P_2, P_3 and have CPU bursts with the above lengths.
- The Gantt chart of the resulting schedule is:

- Waiting times: $P_1 = 0$, $P_2 = 24$, $P_3 = 27$
- Average waiting time = $(0 + 24 + 27)/3 = 17$

First Come, First Served (FCFS) Scheduling (Cont.)

Process	Burst duration
P_1	24
P_2	3
P_3	3

- Now suppose that the same processes arrive in the order: P_2, P_3, P_1.
- The Gantt chart for the schedule is:

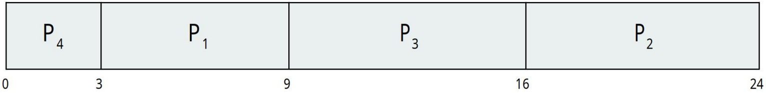
- Waiting times: $P_1 = 6$, $P_2 = 0$, $P_3 = 3$
- Average waiting time = $(6 + 0 + 3)/3 = 3$. Much better than before!
- **Convoy effect** — short processes remain stuck behind long process
  - Result in lower hardware resources utilization in the case of one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its *next CPU burst*
- Use these lengths to schedule processes in reverse burst length order (**shortest burst first**)
- Nonpreemptive

Example:

Process	Burst duration
P_1	6
P_2	8
P_3	7
P_4	3



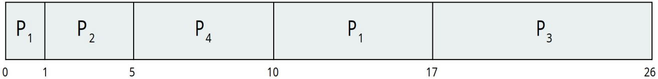
Average waiting time: $(3 + 16 + 9 + 0)/4 = 7$

- SJF is **provably optimal:** it gives minimum average waiting time for a given set of processes
- Problem: how do we determine the length of the next CPU burst? (without knowing the future)
  - Could ask the user
  - Estimate based on past process statistics

Shortest-Remaining-Time-First (SRT) Scheduling

- Preemptive variant of SJF
- Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJF algorithm.
  - Can result in preempting the currently running process
- Is SRT “more optimal” (now that we allow preemption) than SJF in terms of the minimum average waiting time for a given set of processes?

Process	Arrival time	Burst duration
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



- Note how P_1 is preempted by P_2 upon its arrival
- Average waiting time (SRT): $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5$
- Average waiting time with SJF would have been: 7.75

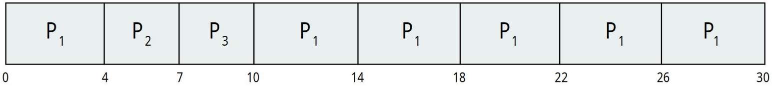
Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds.
- After this time has elapsed, the **process is preempted** and added to the end of the ready queue.
  - Timer interrupts occur every quantum to trigger preemption + scheduling of next process.
- If there are n processes in the ready queue and the time quantum is q, then:
  - Each process gets $1/n$ of the CPU time, in chunks of at most q time units at once.
  - No process waits more than $(n-1)q$ time units.
- Performances depend heavily on q
  - q too large → degenerates to FCFS scheduling
  - q too small → lot of time wasted in context switches, instead of process work

Round Robin (RR) Scheduling — Example

Process	Burst duration
P_1	24
P_2	3
P_3	3

- With $q = 4$ the schedule is:



- Note how P_1 keeps being rescheduled after the termination of P_2 and P_3

- Typical performances: higher average turnaround time than SJF, but better response time
- q should be large compared to context switch time. Typical figures:
  - $q \in$ 10-100 ms
  - context switch < 10 µs

Priority Scheduling

- General class of scheduling policies
- A **priority number** (integer) is associated with each process
- CPU allocated to the **process with the highest priority**
  - Conventionally: smallest integer $\rightarrow$ highest priority
- Can be preemptive or nonpreemptive

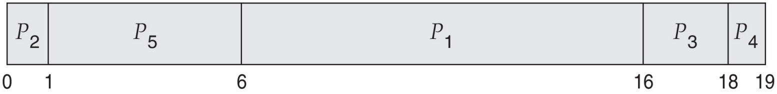
- Note: SJF is an instance of priority scheduling, where priority is the inverse of next CPU burst time

- Problem: **Starvation** — low priority processes may never execute
- Solution: **Aging** — as time progresses increase the priority of a waiting process; eventually it will become “important enough” to be scheduled

Priority Scheduling — Example

Process	Burst duration	Priority
P_1	10	3
P_2	1	1 (highest)
P_3	2	4
P_4	1	5 (lowest)
P_5	5	2

- Resulting schedule with nonpreemptive priority scheduling:



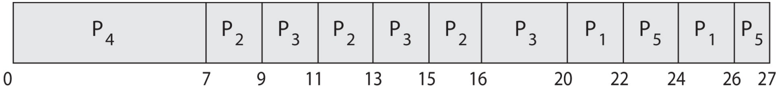
- Average waiting time: $(0+1+6+16+18)/5 = 8.2$

Priority Scheduling with Round-Robin

- Run the process with the highest priority. Processes with the same priority run round-robin.
- Example:

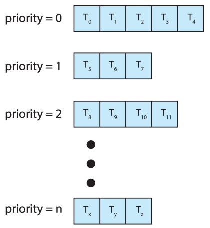
Process	Burst duration	Priority
P_1	4	3 (lowest, ex aequo)
P_2	5	2
P_3	8	2
P_4	7	1 (highest)
P_5	3	3 (lowest, ex aequo)

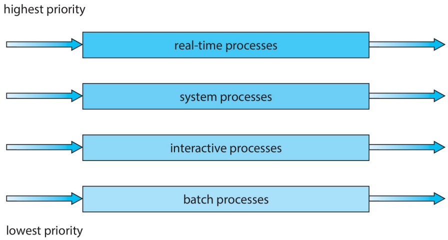
- Schedule for $q = 2$ with RR preemption at quantum expiration:



Multilevel Queue Scheduling

- The ready queue consists of multiple queues
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues
- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!
- Queues organized either by fixed priority (left) or by process type (right):



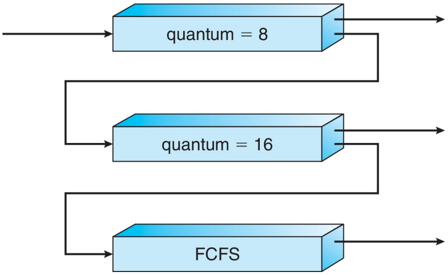


Multilevel Feedback Queue Scheduling

- More general version of multilevel queue scheduling
- Now processes can **move between queues**
- Parameters are the same of multilevel queue scheduling (cf. previous slide), *plus*:
  - Method used to determine when to upgrade a process (to a higher-priority queue)
  - Method used to determine when to demote a process (to a lower-priority queue)
- The most general and most complex scheduling algorithm

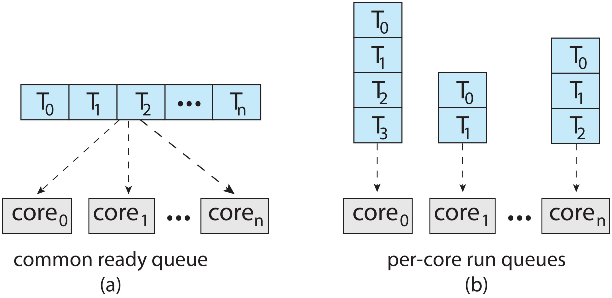
Multilevel Feedback Queue Scheduling — Example

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



SMP Scheduling

- CPU scheduling becomes more complex when multiple CPUs/cores are available
- Many different architectures to consider
  - Multicore CPUs, multithreaded cores, NUMA systems, heterogeneous multiprocessing
- Let's look at a simple and common case: **symmetric multiprocessing (SMP)** scheduling, where each processor is self scheduling.
- Ready threads may be in a (a) common queue or (b) per-processor queues:



SMP Scheduling — Load Balancing

- With SMP, need to **keep all CPUs loaded** for efficiency
- **Load balancing** attempts to keep workload evenly distributed. Two approaches:
  - **Push migration** – periodic task checks load on each processor, and if needed moves tasks from overloaded CPU to other CPUs
  - **Pull migration** – idle processors can pull waiting task from a busy processor

Processor Affinity

- When a thread has been running on one processor, the **cache** contents of that processor stores the memory accessed by that thread.
- We refer to this as a thread having affinity for a processor (i.e., "**processor affinity**")
- **Load balancing affects processor affinity** as when a thread moves from one processor to another, it *loses the contents of what it cached* of the processor it was moved off of. Solutions:
  - **Soft affinity** – the OS attempts to keep a thread running on the same processor, but no guarantees.
  - **Hard affinity** – allows a process to specify a fixed set of processors it may run on.

More moving parts that the scheduler should take into account for its decisions!

Case Study — Linux Scheduling

Linux Scheduling through v2.5

- Prior to kernel version 2.5, ran variation of **historical UNIX scheduling** algorithm
  - Round Robin with priority and aging
  - Problem: $O(n)$ complexity for selecting next task to run
- Version 2.5 moved to the so-called **O(1) scheduler**
  - Preemptive, priority based
  - Two priority ranges: **time-sharing** (normal) and real-time
  - **Real-time** range from 0 to 99; normal range from 100 to 139
  - `nice(1)` (see man page) value from -20 to 19 added to the priority → allow manual tuning
  - Result into a global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task runnable as long as time left in time slice (active)
  - If no time left (expired), not runnable until all other tasks use their slices
  - All runnable tasks tracked in **per-CPU run queue** data structure
- Worked well, but poor response times for interactive processes

Linux Completely Fair Scheduler (CFS)

- Starting with Linux 2.6.23: **completely fair scheduler (CFS)**[2]
- Configurable **scheduling classes**
  - Two predefined scheduling classes—real-time and default—others can be added
  - Each task has a specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on **proportion of CPU time**
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency**: interval of time during which *task should run at least once*
  - Target latency can increase if, e.g., number of active tasks increases
- CFS scheduler maintains per-task **virtual run time** in variable `vruntime`
  - Try it out: `cat /proc/<PID>/sched` and look for “vruntime”
  - Associated with decay factor based on priority of task: lower priority has higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler **picks task with lowest virtual run time**

[2] implemented in kernel/sched/fair.c

Scheduling Evaluation

Deterministic Modeling

- How to select CPU-scheduling policy/algorithm for an OS?
  - Question relevant for both OS implementers and *users*, because in some cases you can adapt/change scheduling policies
- Determine criteria, then evaluate algorithms

- One approach is **deterministic modeling**
  - Type of analytic evaluation
  - Takes a **predetermined workload** and analytically evaluate the performance of each algorithm on it
  - Example: consider the following 5 processes arriving at time 0:

Process	Burst duration
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

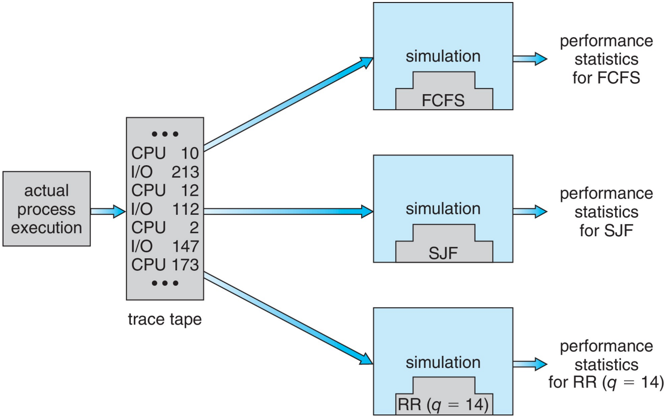
- For each algorithm, calculate the average waiting time
  - e.g., FCFS is 28, SJF 13, RR (q=10) 23
- Pro: simple and fast
- Con: requires exact numbers for input, and is relevant only to those (or very similar) inputs

Queueing Models

- Describes the **arrival of processes**, and CPU and I/O bursts probabilistically (using queueing theory)
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc.
- Computer system described as network of servers, each with queue of waiting processes
  - Requires knowing arrival rates and **service rates**
  - Computes utilization, average queue length, average wait time, etc.

Simulations

- Queueing models are limited
- **Simulations** can be more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Simulation inputs gathered via:
    1. Random number generator according to probabilities
    2. Distributions defined mathematically or empirically
    3. **Traces** of real events recorded from real systems



Implementation

- Even simulations have limited accuracy
- Just implement (**code it up**) new scheduler policy and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
  - Or APIs to modify priorities
- But again environments vary
  - Extrapolating from one system/workload to another is risky

Reading List

You should study on books, not slides! Reading material for this lecture is:

- Silberschatz, Galvin, Gagne. Operating System Concepts, Tenth Edition:
  - Chapter 4: Threads & Concurrency
  - Chapter 5: CPU Scheduling

Credits:

- Some of the material in these slides is reused (with modifications) from the official slides of the book Operating System Concepts, Tenth Edition, as permitted by their copyright note.