Overview

Summary 4: p253 Summary 5: p317 Summary 6: p361 Summary 7: p400

Chapter 4 Review

Benefits of Threads (Slide 4.9)

- Responsiveness;
- Resources Sharing;
- Economy;
- Scalability.

Multicore Programming Challenges (Slide 4.10)

- Dividing activities;
- Balance;
- Data splitting;d
- Data dependency;
- Testing and debugging.

Parallelism & Concurrency (Slide 4.10)

- Parallelism implies a system can perform more than one task simultaneously;
- Concurrency supports more than one task making progress (single processor, scheduler providing concurrency).

Types of Parallelism (Slide 4.12)

- **Data parallelism**: distributes subsets of the same data across multiple cores, same operation on each.
- **Task parallelism**: distributes threads across cores, each thread performing unique operation.

Amdahl's Law (Slide 4.14)

$$speedup \leq \frac{1}{S + \frac{1-S}{N}}$$

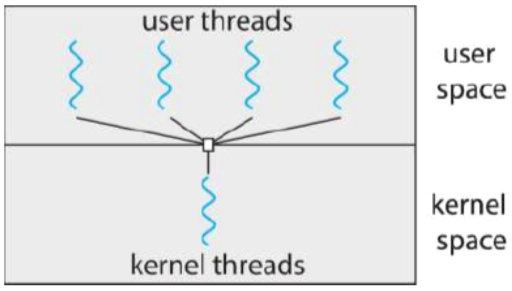
where S: serial portion; N: core numbers.

User Thread & Kernel Thread (Slide 4.16)

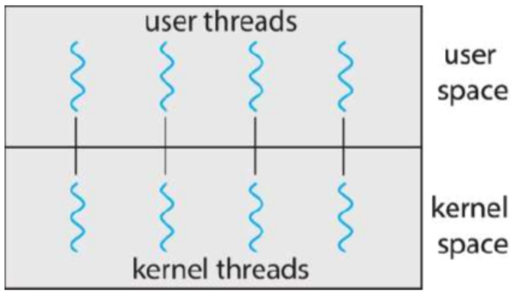
- User threads: management done by user-level threads library;
- Kernel threads: supported by the Kernel.

Multithreading Models

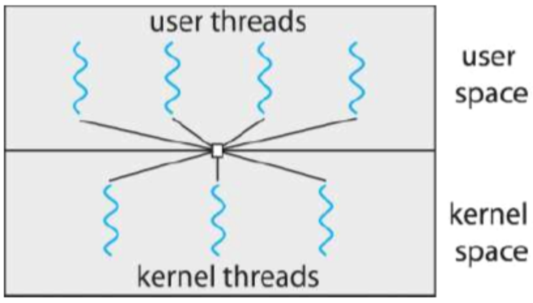
- **Many-to-one** (Slide 4.19): Many user-level threads mapped to single kernel thread;



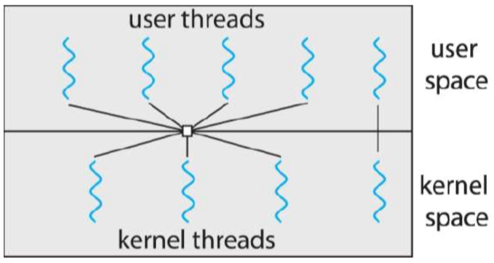
- **One-to-one** (Slide 4.20): Each user-level thread maps to kernel thread;



- **Many-to-many** (Slide 4.21): Allows many user level threads to be mapped to many kernel threads;



- **Two-level** (Slide 4.22): like Many-to-many model, except it allows a user thread to be bound to kernel thread.



Thread Library (Slide 4.23): provides programmer with API for creating and mangeing threads. Two main implementing ways:

- Library entirely in user space;
- Kernel-level library supported by the OS.

Pthreads Example (Slide 4.25).

Implicit Threading (Slide 4.35): involves identifying tasks - not threads, and creation and management of threads done by compilers and run-time libraries rather than programmers. Five methods explored:

- **Thread Pools** (Slide 4.36): create a number of threads in a pool where they await work.

  Advantage of Thread Pools (Slide 4.36):

    - Usually sightly faster to service a request with an existing thread than create a new thread;
    - Allows the number of threads in the applications to be bound to the size of the pool;
    - Separating task to be performed from mechanics of creating tasks allows different strategies for running task (i.e., task could be scheduled to run periodically).
- **Fork-Join** (Slide 4.39): multiple threads (tasks) are forked, and then joined.
- OpenMP (Slide 4.45): set of compiler directives and API for C, C++, FORTRAN, identifies *parallel regions* in source code.
- Grand Central Dispatch (Slide 4.47): Allows identifications of a parallel sections using `^{}` (apple design).
- Intel Threading Building Blocks (TBB) (Slide 4.50): using `parallel_for` instruction.

Semantics of `fork()` and `exec()` (Slide 4.52)

- In Linux, `fork()` usually duplicate the calling thread; but in UNIX, there may exist two versions.
- `exec()` replace the running process including all threads.

Signals & Signal Handling (Slide 4.53)

- Signal: to notify a process that a particular event has occurred in UNIX system.
- **Signal handler**: default handler (by default), user-defined signal handler (can override default).
- **Problem** (Slide 4.54): deliver signal to which thread? Four methods:
    - to which it applies;
    - to every thread in the process;
    - to certain threads in the process;
    - assign a specific thread to receive all signals of the process.

Thread Cancellation (Slide 4.55)

- **Target thread**: thread to be cancelled;
- **General approaches**:
    - Asynchronous cancellation: terminates immediately;
    - Deferred cancellation: allows the target thread to periodically check if it should be cancelled.

Thread-Local Storage (TLS) (Slide 4.58) allows each thread to have the own copy of data.

- Own stack & register spaces.

Lightweight process (LWP) (Slide 4.59) an intermediate data structure between user and kernel threads.

Tasks in Linux (Slide 4.64) Linux does not distinguish between processes and therads, and it refers to each as a task. `clone()` system call can be used to create tasks that behave either more like processes or more like threads (depends on the arguments).

Chapter 5 Review

CPU-I/O Burst Cycle (Slide 5.6): process execution consists of a cycle of CPU execution and I/O wait. CPU burst followed by I/O burst.

CPU Scheduler (Slide 5.8) selects from the processes in ready queue, and allocates a CPU core to one of them.

Non-preemptive & preemptive (Slide 5.8): scheduler under the following rules in non-preemptive; otherwise it is preemptive. Scheduler make decisions when a process:

1. switches from running to waiting state;
2. switches from running to ready state;
3. switches from waiting to ready state;
4. Terminates.

Dispatcher module (Slide 5.9) gives control of the CPU to the process selected by the short-term scheduler; this involves:

- switching context;
- switching to user mode;
- jumping to the proper location in the user program to restart the program.

and **dispatch latency** is the time it takes for the dispatcher to stop one process and start another running.

Scheduling Criteria (Slide 5.10, Slide 5.11)

- CPU utilization: max; (not-often-used, affected by hardware)
- Throughput: max; (not-often-used, affected by hardware)
- Turnaround time: min;
- Waiting time: min;
- Response time: min.

Scheduling algorithms

- **FCFS** (Slide 5.12) (default: non-preemptive);
- **SJF** (Slide 5.14) (default: non-preemptive): shortest average waiting time;
- **SRTF / SRF** (Slide 5.16), shortest-remaining-time-first (default: preemptive), a preemptive version of SJF;
- **RR** (Slide 5.20) (default: preemptive);
- **Priority** (Slide 5.24) (default: preemptive).
  - May cause **starvation**;
  - Use **aging** method to solve it.
- **Multilevel Queue** (Slide 5.27) (Priority + (RR / other algorithms));
  - Priority from high to low: real-time processes; system processes; interactive processes; batch processes.
- **Multilevel Feedback Queue** (Slide 5.29): similar to Multilevel Queue, except that a process can move between various queues. A dynamic version of Multilevel Queue.

Thread Scheduling (Slide 5.31)

- User-level thread schedulers often in LWP, known as **Process-Contention Scope (PCS)** since the scheduling competition is within the process; and it is typically done via priority set by programmer;
- Kernel-level thread scheduled onto available CPU is **System-Contention Scope (SCS)** - competition among all threads in system.

Multiple-Processor Scheduling (Slide 5.35)

- Multiprocessor architectures:

    - Multicore CPUs;
    - Multithreaded cores;
    - NUMA systems;
    - Heterogeneous multiprocessing.
- **Load Balancing** (Slide 5.41) attempts to keep workload evenly distributed;
    - **Push migration**: periodic task checks load on each processor, and if found, pushes task from overloaded CPU to other CPUs;
    - **Pull migration**: idle processors pulls waiting task from busy processor.
- **Processor Affinity** (Slide 5.42)
    - **Soft affinity**: attempts to keep a thread running on the same processor, but no guarantees;
    - **Hard affinity**: allows a process to specify a set of processors it may run on.

Real-time CPU Scheduling (Slide 5.44)

- **Soft real-time systems**: critical real-time tasks have the highest priority, but no guarantees as to when tasks will be scheduled; **Priority-based** scheduling algorithm.
- **Hard real-time systems**: task must be serviced by its deadline; **EDF** (earliest-deadline-first) scheduling algorithm (assigns priority according to the deadline).
- **Event latency** (Slide 5.45): interrupt latency & dispatch latency.
- **Rate monotonic** (Slide 5.49) assigns the priority based on the inverse of the period of process; longer period - lower priority; shorter period - higher priority.
- **Rate monotonic real-time scheduling algorithms**: use rate-monotonic method, priority-based, preemptive.
- **Proportional Share Scheduling** (Slide 5.52) T shares are allocated among all processes in the system; an application receives N shares where $N < T$, and this ensures each application will receive N/T of the total processor time (Divide the processor time into T pieces and allocate them).

Scheduler Examples Linux, Windows, Solaris (Page 251 textbook, or page 318 in PDF)

Evaluate a CPU Scheduling Algorithm (Slide 5.70)

- Modeling;
- Simulations;

Little's Formula (Slide 5.73)

$$n = \lambda \times W$$

where n is average queue length; λ is average arrival rate; and W is average waiting time in queue.

Chapter 6 Review

Critical Section (Slide 6.11): Process may be changing common variables, updating table, writing file, etc; when one process is in critical section, no other may be in its critical section.

- **Entry section**: asks permission to enter critical section;
- **Exit section**: follows critical section with exit section;
- **Remainder section**: the rest of section.

Critical Section Solution (Slide 6.13)co

- **Mutual Exclusion**: only one process can be executing in its critical section;

- **Progress**: if not process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of processes that will enter the critical section next cannot be postponed indefinitely;
- **Bounded Waiting**: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted;
    - Assume that each process executes at a nonzero speed;
    - No assumption concerning relative speed of n processes.

Critical-Section Handling in OS (Slide 6.14)

- **Preemptive**: allows preemption of process when running in kernel mode;
- **Non-preemptive**: runs until exists kernel mode, blocks, or voluntarily yields CPU (essentially free of race conditions in kernel).

Peterson's Solution (Slide 6.15)

```
int turn;
bool flag[2];
```

- Not useful (Slide 6.18)

Synchronization Hardware (Slide 6.21)

- Uniprocessors: could disable interrupts; too inefficient on multiprocessor systems;
- **Three hardware support**:
    - **Memory barriers** (Slide 6.22): an instruction that forces any change in memory to be propagated (made visible) to all other processors.
        - Strongly ordered memory model
        - weakly ordered memory model
    - **Hardware instructions** (Slide 6.24): test-and-set instruction, compare-and-swap instruction.
    - **Atomic variables** (Slide 6.30): provides atomic (uninterruptible) updates on basic data types.

Mutex Lock (spinlock) (Slide 6.32): `acquire()` and `release()`, requires **busy waiting**.

- Implementation (Slide 6.34).

Semaphore (Slide 6.35): can only be accessed via two indivisible (atomic) operations: `wait()` and `signal()`.

- **Counting semaphore**: integer value can range over an unrestricted domain;
- **Binary semaphore**: integer value can range only between 0 and 1; same as mutex lock.
- Can be implemented with no busy waiting (Slide 6.38) (use a list):
    - **block**: place the process invoking the operation on the appropriate waiting queue;
    - **wakeup**: remove one of processes in the waiting queue and place it in the ready queue.

Monitors (Slide 6.41) a high-level abstraction that provides a convenient and effective mechanism. Abstract data type, internal variables only accessible by code within the procedure. Only one process may be active within monitor at a time.

Condition Variables: allows processes to be blocked inside the monitor (if data is not ready).

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, then there are two options:
    - **Signal and wait**: P waits until Q either leaves monitor of it waits for another condition (implement in Slide 6.47)
    - **Signal and continue**: Q waits until P either leaves the monitor or it waits for another condition.
- (Slide 6.49) Conditional-wait construct of form `x.wait(c)` where `c` is priority number, process with lowest number (highest priority) is scheduled next.

Liveness (Slide 6.52) refers to a set of properties that a system must satisfy to ensure processes make progress.

Deadlock (Slide 6.53): two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- Starvation & Priority Inversion (Slide 6.54).

Chapter 7 Review

Bounded-Buffer Problem (Slide 7.6) Solution:

- Semaphore `mutex` to prevent race condition;
- Semaphore `full` to prevent reading when empty;
- Semaphore `empty` to prevent writing when full.

Readers-Writers Problem (Slide 7.9) Solution:

- Semaphore `rw_mutex` to prevent reading & writing at the same time;
- Integer `read_count` to record how many readers are in there.
- Semaphore `mutex` to prevent race condition on `read_count`.

Dining-Philosophers Problem (Slide 7.13) Solution: monitor; an conditional variable for each philosopher (Slide 7.15).

Applications (Java / POSIX / ...)