Operating Systems Midterm

2017-02-22

2017-03-01

- CPU, memory, and devices are the 3 major components of a computing machine.
- Direct memory access (DMA) is a way for devices of bypassing CPU to access memory, without increasing the loading of the CPU.
- CPU executes the same process until receiving an **interrupt** generated by either a device or the kernel. On receiving an interrupt, CPU would save the status of the currently-running process before moving to another process.
- A process can be in either **user mode** or **kernel mode**.
- On Linux, when a user call a library function, e.g. `open()`, a corresponding system call would be triggered, in this case, `sys_open()`.
- Microsoft Disk Operating System (MS-DOS) has a drawback that the entire memory space is occupied by a single process.
- Real-time OS is aimed at executing a task within a time limit with precise scheduling mechanism.
- Kernel is encapsulated by layers of function calls, with the innermost ones dealing with the most fundamental operation.
- Kernels fall into 2 major categories: **monolithic** and **microkernel**. In the case of microkernel, the kernel only retains the system calls involving instructions that must be done by the hardware, including the following 3 situations:
    - CPU clock
    - Process/thread management
    - Synchronization
- Linux modules are the programs compiled on user memory space but executed in the kernel.
- IO interleaving, or **spooling**, is the most influential factor determining the efficiency of an operating system. The type of a kernel, i.e. either monolithic or a microkernel, has less influence.
- Machines can fall into 3 major categories: non-virtual, **paravirtual**, and **full virtual** machine. A virtual machine makes use of time to simulate that every operating system on it has complete control over the memory and devices. But virtual machine can never guarantee how long it takes to run a program.

- [x] Homework: compile Linux kernel on Ubuntu OS
- Solutions: kernel.sh

2017-03-08

- The kernel mode in virtual machine is a user mode in actual, therefore, some instructions are not allowed to execute. But this problem can be solved by additional handling.
- **Container** in Solaris 10 is a light-weighted virtualization.
- **VMware architecture** leaves hardware handling to the host operating system. Whereas **Xen architecture** handles hardware operations by itself, therefore, it is potentially more efficient than VMware.
- Debugging an operating system can be realized through virtualization.
- **Spooling** problems: IO can never catch up with the speed of CPU clock. If CPU has enough buffer, the efficiency of IO can be partially optimized.
- **Batch system** can save time for *context switching*. A batch is not processed until CPU is available.
- Why should text segment grow from the bottom (memory address = 0)?
    - Faster access to the memory address by bit shifting without an offset.
    - Save power to run the process since lower memory addresses have more 0.
- A **zombie** process is designed for its parent process to wait for. If there is no such a state, the termination status of a process will be lost.
- The ways to run system calls, function calls, and context switching are different.
- IO queue is designed to be first in first out (FIFO), and, therefore, not a real-time design.
- Several scenarios would lead to a context switching: e.g expired time slice, IO request, forking a child, waiting for an interrupt, etc.
- Time scheduling can be **long-term, medium term**, and **short term**.
- **Producer-consumer problem** is to discuss how a message generator and receiver exchange information.

2017-03-15

- A **thread** is a basic unit of CPU utilization; it comprises a *thread ID*, a *program counter*, a *register set*, and a *stack*. It shares with other threads belonging to the same process its *code section*, *data section*, and other operating-system resources, such as *open files* and *signals*.
- The benefits of a multi-threaded process includes:
    - *Responsiveness*: a program to continue running even if part of it is blocked or is performing a

lengthy operation, i.e. more efficient in IO interleaving, or **spooling**.
    - *Resource sharing*: threads share the memory and the resources of the process to which they belong by default.
    - *Economy*: it is more economical to create and context-switch threads than processes.
    - *Scalability*: threads can be running in parallel on different processing cores.
- Each CPU core will be assigned to execute only a thread at any time.
- **Parallelism** v.s. **Concurrency**:
    - A system is *parallel* if it can perform more than one task simultaneously.
    - A *concurrent* system supports more than one task by allowing all the tasks to make progress.
    - CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes in the system.
- **Amdahl's Law**: speedup ≤ $\frac{1}{S+(1-S)/N}$, where S is the *portion* of the application that must be performed serially on a system, N is the number of processing cores
    - Identifies potential performance gains from adding additional computing cores to an application that has both serial (nonparallel) and parallel components.
    - Implies the serial portion of an application can have a disproportionate effect on the performance we gain by adding additional computing cores.
- Challenges in programming for multicore systems:
    - *Identifying tasks*.
    - *Balance*: the tasks perform equal work of equal value.
    - *Data splitting*.
    - *Data dependency*: the execution of the tasks has to be synchronized to accommodate the data dependency.
    - *Testing and debugging*.
- Two types of parallelism: **data parallelism** and **task parallelism**.
    - **Data parallelism**: distributing subsets of the same data across multiple computing cores.
    - **Task parallelism**: distributing not data but tasks (threads) across multiple computing cores.
    - In most instances, applications use a *hybrid* of these two strategies.
- **User threads** v.s. **kernel threads**
    - **User threads** are supported above the kernel (e.g. by a threads library) and are managed without kernel support.
    - **Kernel threads** are supported and managed directly by the operating system.
- Thread models: **many-to-one model**, **one-to-one model**, **many-to-many model**, and **two-level model**.

- **Many-to-One Model**:
    - Maps many user-level threads to one kernel thread.
    - Managed by the *thread library* in user space, so it is *efficient*.
    - *Shortcomings*:
        - The entire process will block if a thread makes a blocking system call.
        - Only *one* thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.
- **One-to-One Model**:
    - Maps each user thread to a kernel thread.
    - Run in parallel on multiprocessors.
    - *Shortcomings*:
        - Overhead of creating kernel threads can burden the performance of an application.
        - Restrict the number of threads.
- **Many-to-Many Model**:
    - Multiplexes many user-level threads to a smaller or equal number of kernel threads.
    - Suffers from neither of the above shortcomings.
- **Two-level model**:
    - Not only multiplexes user threads but also allows a user-level thread to be bound to a kernel thread.
- *Two* primary ways of implementing thread libraries: entirely in the user space, or a kernel-level library.
- *Two* general strategies for creating multiple threads: **asynchronous threading** and **synchronous threading**:
- **Asynchronous threading**:
    - Once the parent creates a child thread, the parent resumes its execution.
    - Each thread runs independently of every other thread.
    - *Little* data sharing between threads.
- **Synchronous threading**:
    - **Fork-join** strategy: the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes.
    - *Significant* data sharing among threads.
- **Implicit threading**: transfer the creation and management of threading from developer to compilers and run-time library. *Three* alternative approaches are *thread pool*, *OpenMP*, *Grand Central Dispatch (GCD)*.

- **Thread Pools**:
    - Create a number of threads in a pool where they await work.
    - Servicing a request with an existing thread is faster than waiting to create a thread.
    - A thread pool limits the number of threads that exist at any one point.
    - Easier to use different strategies for running the task, e.g. *delay*, or *periodic*.
    - *Dynamically* adjusting the number of threads in the pool consumes even less memory.
- **OpenMP** identifies parallel regions as blocks of code that may run in parallel.
- **Grand Central Dispatch (GCD)** identifies a self-contained unit of work specified by a caret .
- Threading issues:
    - The amount of time required to create the thread.
    - Unlimited threads could exhaust system resources.
- Does the new process duplicate all threads, or is the new process single-threaded?
    - Some operating systems implemented both.
    - Depends on if a child process invokes `exec()` immediately after forking.
- Signals is either *synchronous* or *asynchronous*:
    - **Synchronous**: signals delivered to the same process that performed the operation that caused the signal.
    - **Asynchronous**: signal generated by an event external to a running process.
- Signal handling for threads
    - A signal is generated by the occurrence of a particular event.
    - The signal is always *delivered* to a process.
    - Once delivered, the signal must be *handled*, either by a default signal handler or a user-defined signal handler.
    - *Synchronous* signals need to be delivered to the thread causing the signal and not to other threads in the process. However, the situation with *asynchronous* signals is not as clear.
    - An *asynchronous* signal may be delivered only to those threads that are not blocking it. However, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it.
    - `pthread_kill()` allows a signal to be delivered to a specified thread (tid).
- **Asynchronous procedure calls (APCs)** is Windows' version of signals, which is delivered directly to a particular *thread* rather than a process.
- **Thread cancellation**
    - A thread to be canceled is referred to as the **target thread**.
    - **Asynchronous cancellation**:

        - One thread *immediately* terminates the target thread.
        - Canceling a thread *asynchronously* may not free a necessary system-wide resource.
    - **Deferred cancellation**:
        - The target thread *periodically* checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.
        - Actual cancellation depends on how the target thread is set up to handle the request.
- **Thread-local storage (TLS)** v.s. local variables:
    - Local variables are visible only during a single function invocation.
    - TLS data are visible across function invocations, more like a `static` variable, but unique to each thread.
- Many systems place an intermediate data structure between the user and kernel threads, known as a **lightweight process (LWP)**, or **virtual processor**.
- **Scheduler activation**, or **upcall**: a scheme for communication between the *kernel* and the *thread library*.
    - The kernel provides an application with a set of *virtual processors (LWPs)*, and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events.
    - Upcalls are handled by the *thread library* with an **upcall handler**, and upcall handlers must run on a *virtual processor*.
- When an application thread is about to block:
    - The kernel makes an *upcall* to the application (thread library) informing it that a thread is about to block and identifying the specific thread.
    - The kernel then allocates a new virtual processor to the application.
    - The application runs an *upcall handler* on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.
- When the event that the blocking thread was waiting for occurs:
    - The kernel makes another *upcall* to the application (thread library) informing it that the previously blocked thread is now eligible to run.
    - The kernel then allocates a new virtual processor to the application.
    - The application run the upcall handler to schedule an eligible thread to run on an available virtual processor.

2017-03-23

- Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.
- Process execution consists of a **cycle** of **CPU burst** (CPU execution) and **IO burst** (I/O wait).
- The curve of CPU-burst durations is generally characterized as *exponential* or *hyperexponential*.
- 5 + 1 states of a thread: *new*, *ready*, *running*, *waiting*, *terminated*, and *zombie*.
- **CPU scheduler**, or **short-term scheduler**, selects a process from the processes in memory (*ready queue*) that are ready to execute and allocates the CPU to that process.
- A ready queue can be implemented as a *FIFO queue*, a *priority queue*, a *tree*, or simply an *unordered linked list*.
- The records in the queues are generally **process control blocks (PCBs)** of the processes.
- CPU scheduling decisions may take place when a process:
    - Switches from *running* to *waiting* state (**nonpreemptive**).
    - Switches from *running* to *ready* state (**preemptive**).
    - Switches from *waiting* to *ready* state (**preemptive**).
    - Terminates (**nonpreemptive**).
- **Nonpreemptive scheduling**, or **cooperative scheduling**:
    - Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by *switching to the waiting state* or by *terminating*.
    - *Preemptive scheduling* does *not* require a special hardware (e.g. a timer).
- **Preemptive scheduling**:
    - Can result in *race conditions* when (1) data are shared among several processes, (2) a system call updating kernel data is preempted by the kernel (or the device driver) needing to read or modify the same data.
- *Solution* for *race conditions*:
    - By waiting either for a system call to complete or for an I/O block to take place before doing a *context switch*.
    - The kernel will *not* preempt a process while the kernel data structures are in an *inconsistent* state.
    - The sections of code affected by interrupts must be guarded from simultaneous use.
    - *Critical* sections of code may *disable* interrupts at entry and *reenable* interrupts at exit.
    - The *solution* is a *poor* one for supporting *real-time* computing where tasks must complete execution within a given time frame.
- **Dispatcher** is the module that gives control of the CPU to the process selected by the *short-term scheduler*.

    - Switching context → Switching to user mode → Jumping to the proper location in the user program to restart that program.
- **Dispatch latency** is the time it takes for the *dispatcher* to stop one process and start another running.
- **Scheduling criteria**:
    - **CPU utilization**: keep the CPU as busy as possible.
    - **Throughput**: the number of processes that are completed per time unit.
    - **Turnaround time**: from the time of submission of a request to the time of completion. The turnaround time is generally limited by the speed of the output device.
    - **Waiting time**: the sum of the periods spent waiting in the ready queue.
    - **Response time**: from the submission of a request until the first response is produced.
- **Gantt chart**: a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes.
- In the following discussion of scheduling algorithms, we consider only *one CPU burst (in milliseconds) per process*. Our measure of comparison is the *average waiting time*.
- **First-come, first-served (FCFS) scheduling**:
    - Easily managed with a *FIFO queue*, and always *nonpreemptive*.
    - The *average waiting time* is generally not minimal and may vary substantially.
    - **Convoy effect**: all the other processes wait for the one big process to get off the CPU.
- **Shortest-job-first (SJF) scheduling**:
    - If the next CPU bursts of two processes are the same, *FCFS* scheduling is used to break the tie.
    - Better called **shortest-next-CPU-burst** algorithm.
    - Provably optimal average waiting time by *proof by contradiction*: assume that there is a shortest job first (SJF) schedule S, where the optimal schedule is a non-SJF schedule S', and then prove that the average waiting time of S is less than that of S'.
    - Can be either *preemptive* or *nonpreemptive*.
    - *Nonpreemptive*: checks priority only when a job ends.
    - *Preemptive*: checks priority when a job ends or when a new job comes, also known as **shortest-remaining-time-first (SRTF)** scheduling.
    - Used frequently in *long-term* scheduling.
    - With *short-term* scheduling, there is no way to know the length of the next CPU burst.
    - The next CPU burst is generally predicted with an **exponential average**: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$, where t_n is the length of the most recent *real* CPU burst, and τ_n is the length of the most recent *predicted* CPU burst and stores the past history.

- **Priority scheduling**:
  - An *SJF* algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst.
  - Equal-priority processes are scheduled in *FCFS* order.
  - Priorities can be defined either *internally* or *externally*.
  - Can be either *preemptive* or *nonpreemptive*.
  - **Starvation**: low priority processes may never execute.
  - **Aging**: increase the priority of processes that wait in the system for a long time.
- **Round-robin (RR) scheduling**:
  - Similar to *FCFS* scheduling, but *preemption* is added to enable the system to switch between processes every a small unit of time, called a **time quantum** or **time slice**.
  - Each process must wait no longer than $(n-1) \times q$ time units for the first CPU burst (i.e. *short average response time*), where there are n processes in the ready queue and the time quantum is q.
  - The ready queue is treated as a *circular queue*.
  - If the time quantum q is:
    - *large*: the same as the FCFS policy.
    - *small*: result in a large number of context switches.
  - The *average waiting time* under the RR policy is often *long*.
  - Typically, RR policy has *longer average turnaround time* than SJF, but *shorter average response time*.
  - *Turnaround time* depends on the size of the *time quantum*. The *average turnaround time* can be improved if most processes finish their next CPU burst in a single time quantum.
  - A rule of thumb is that *80%* of the CPU bursts should be shorter than the time quantum.
- **Multilevel queue scheduling**:
  - Ready queue is partitioned into separate queues: **foreground (interactive)** v.s. **background (batch)**.
  - Processes are *permanently* assigned to one queue. *Inflexible*!
  - Each queue has its own scheduling algorithm, e.g. the *foreground* queue scheduled by RR, and the *background* queue scheduled by FCFS.
  - Scheduling among the queues is commonly implemented as *fixed-priority preemptive scheduling*, or by *time-slicing* among the queues.
    - *Fixed-priority preemptive scheduling*: risk of starvation.
    - *Time-slicing*: each queue gets a certain percentage of CPU time which it can schedule

amongst its processes.

- **Multilevel feedback queue scheduling**:
  - A process can move between the various queues.
  - Allow *aging* to prevent *starvation*.
  - If a higher priority process does not finish within a specified time quantum, it is moved to the tail of the next lower-priority queue.
- It is *kernel-level* threads, *not* processes, that are being scheduled by the *operating system*.
- **Process-contention scope (PCS)**:
  - The *thread library* schedules *user-level* threads onto an available *LWP*.
  - Competition for the CPU takes place among threads belonging to the same process.
  - On systems implementing the *many-to-one* and *many-to-many* models.
- **System-contention scope (SCS)**:
  - The *operating system* schedules *kernel-level* threads onto a *physical processor*.
  - Competition for the CPU with SCS scheduling takes place among all threads in the system.
  - On systems implementing the *one-to-one* model.
- Multiple processor scheduling:
  - **Asymmetric multiprocessing**: only one processor accesses the system data structures, reducing the need for data sharing.
  - **Symmetric multiprocessing (SMP)**: each processor is self-scheduling. All processes in common ready queue, or each has its own private queue of ready processes.
- **Processor affinity**: a process has an affinity for the processor on which it is currently running to avoid the high cost of *invalidating* and *repopulating* caches between processors.
  - **Soft affinity**: the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors.
  - **Hard affinity**: allows a process to specify a subset of processors on which it may run, e.g. `sched_setaffinity()` on Linux.
- **Non-uniform memory access (NUMA)**: a CPU has faster access to some parts of main memory (i.e. the board where that CPU resides) than to other parts.
- **Load balancing**:
  - Attempts to keep the workload evenly distributed across all processors in an SMP system.
  - Often only necessary on systems where each processor has its own private queue, rather than on systems with a common run queue.
  - Often counteracts the benefits of *processor affinity*.
  - Two general approaches: **push migration** v.s. **pull migration**.

    - **Push migration**: a specific task periodically checks the load on each processor and evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.
    - **Pull migration**: an idle processor pulls a waiting task from a busy processor.
- Multicore processor scheduling:
    - **Memory stall**: when a processor accesses memory, it spends a significant amount of time waiting for the data to become available.
    - *Multithreaded* multicore processor: **coarse-grained** v.s. **fine-grained** multithreading.
    - *First level scheduling*: the operating system chooses which *software thread* (i.e. kernel-level thread) to run on each *hardware thread* (i.e. physical processor).
    - *Second level scheduling*: each core of a processor decides which hardware thread to run.
- **Real-time system (RTS)** scheduling:
    - **Soft real-time**: provides no guarantee as to when a critical real-time process will be scheduled; the usefulness of a result degrades gradually after its deadline.
    - **Firm real-time**: infrequent deadline misses are tolerable, but may degrade the system's quality of service.
    - **Hard real-time**: a task must be serviced by its deadline; service after the deadline has expired is regarded as a total system failure.
- Scheduling strategies used by RTS: *minimizing latency*, *priority-based scheduling*, *rate-monotonic scheduling*, *earliest-deadline-first (EDF) scheduling*, and *proportional share scheduling*.
- **Periodic task model**:
    - Tasks that require the CPU at constant intervals (**periods**).
    - The **rate** of a periodic task is $\mathbf{1}/\boldsymbol{p}$, where $\boldsymbol{p}$ is the period.
    - Has high *predictability* and *schedulability*.
    - **Admission-control**: the scheduler either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.
- Minimizing latency:
    - **Event latency**: the amount of time that elapses from when an event occurs to when it is serviced.
    - **Interrupt latency**: the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt.
    - On receiving an interrupt, the operating system must then save the state of the current process before servicing the interrupt using the specific **interrupt service routine (ISR)**.

    - One important factor contributing to *interrupt latency* is the amount of time interrupts may be *disabled* while kernel data structures are being updated.
    - **Dispatch latency**: the amount of time required for the scheduling dispatcher to stop one process and start another.
    - The most effective technique for keeping *dispatch latency* low is to provide *preemptive kernels*.
- **Rate-monotonic scheduling** (*static priority policy with preemption*):
    - Statically assigned a priority inversely based on its period.
    - The CPU utilization of a process P is the ratio of its burst t to its period p, i.e. t/p.
    - *Schedulability test*: the worst-case CPU utilization for scheduling N processes is $N(2^{1/N} - 1)$, e.g. 100% when $N = 1$; 83% when $N = 2$; 69% when $N = \infty$.
    - As long as combined CPU utilization for N processes is less than the worst-case CPU utilization (i.e. pass the schedulability test), they are guaranteed to be schedulable by rate-monotonic scheduling.
    - CPU utilization is bounded, and it is not always possible to fully maximize CPU resources.
    - *Optimal* static priority scheduler in RTS.
- **Earliest-deadline-first (EDF) scheduling** (*dynamic priority policy with preemption*):
    - Dynamically assigns priorities inversely based on the remaining time before deadline.
    - Unlike the rate-monotonic algorithm, EDF scheduling does *not* require that processes be periodic, *nor* must a process require a constant amount of CPU time per burst.
    - *Schedulability test*: the worst-case CPU utilization for scheduling N processes is $\sum_{i=1}^{N} t_i/p_i \leq 1$.
    - As long as combined CPU utilization for N processes is less than 1 (i.e. pass the schedulability test), they are guaranteed to be schedulable by EDF scheduling.
    - CPU utilization is not bounded theoretically, but it is impossible to achieve this level of CPU utilization due to the cost of context switching.
    - *Optimal* dynamic priority scheduler in RTS.
- **Proportional share scheduling**:
    - Actively allocates CPU shares among all applications.
    - Must work in conjunction with an *admission-control policy*.
- Scheduling on Linux:
    - The $O(1)$ scheduler of the kernel *Version 2.5* delivered excellent performance on SMP systems, but led to poor response times for the interactive processes.
    - The $O(\log N)$ scheduler of the kernel since *Version 2.6* is known as **Completely Fair Scheduler (CFS)**.

    - Scheduling in the Linux system is based on **scheduling classes**, two of which are (1) a default scheduling class using the CFS scheduling algorithm and (2) a real-time scheduling class.
    - **Nice value**: lower nice value indicates a higher relative priority.
    - **Targeted latency**: an interval of time during which every runnable task should run at least once.
    - CFS records how long each task has run by maintaining the **virtual run time** of each task using the per-task variable `vruntime`, which is associated with a *decay factor* inversely based on the priority of a task.
    - Each runnable task is placed in a *red-black tree*. The scheduler simply selects the task that has the *smallest* vruntime value, i.e. the cached *leftmost* leave.
- Scheduling algorithm evaluation can be achieved by *deterministic modeling*, *queueing model*, *simulation*, *emulation*, or *implementation*.
- **Deterministic modeling**: takes a particular predetermined workload and defines the performance of each algorithm
- **Little's formula** for **queuing models**: $n = \lambda \cdot W$, where n is the average queue length (excluding the process being serviced), W is the average waiting time in the queue, and λ is the average arrival rate for new processes in the queue.

2017-03-30

- Producer-consumer problem:
    - In-and-out version: (1) *no* race condition; (2) the buffer is *not* fully filled.
    - Counter version: (1) race condition.
- Counter: `counter++` and `counter--` is implemented in machine language as follows: (1) `register=counter` (2) `register=register±1` (3) `counter=register`.
- **Race condition**: several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.
- **Critical-section problem**:
    - To design a protocol that the processes can use to cooperate.
    - Composed of **entry section**, **critical section**, and **exit section**.
- A solution to the critical-section problem must satisfy the following three requirements:
    - **Mutual exclusion**: if a process is executing in its critical section, then no other processes can be executing in their critical sections.
    - **Progress**: if no process is executing in its critical section and some processes wish to enter their critical sections, the selection of processes to enter their critical section cannot be

postponed indefinitely.

  - **Bounded waiting**: there exists a bound, or limit, 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.

- Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**.
- **Peterson's solution**:
  - The two processes have to share two data items: `turn` and `flag[2]`.
  - `turn` indicates whose turn it is to enter the critical section.
  - `flag[2]` indicates if a process is ready to enter the critical section.
  - *Algorithm 1* with only `turn`: fails the *progress* requirement if a process is blocked in remainder section.
  - *Algorithm 2* with only `flag[2]`: fails the *progress* requirement when `flag[0] = flag[1] = true`.
  - *Peterson's solution*: solves the above problems.

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    // critical section
    flag[i] = false;
    // remainder section
} while (true);
```

- Hardware support for synchronization can be achieved through (1) *disabling interrupts*, and (2) **atomic** hardware instructions working in conjunction with **locking**.
- Through *disabling interrupts*:
  - Adopted by many **nonpreemptive kernels**.
  - Disabling interrupts on a multiprocessor can be time consuming.
- Through **atomic** hardware instructions working in conjunction with **locking**:
  - `test_and_set()`: test memory word and set value.
  - `swap()`: swap contents of two memory words.
  - Does *not* satisfy the bounded-waiting requirement.

```
bool test_and_set(bool *lock) {
    bool is_locked = *lock;
    *lock = true;
    return is_locked;
} // an atomic hardware instruction
do {
```

```
    while (test_and_set(&lock));
    // critical section
    lock = false;
    // remainder section
} while (true);
```

```
bool swap(bool *lock, bool *key) {
    bool is_locked = *lock;
    *lock = *key;
    *key = is_locked;
} // an atomic hardware instruction
do {
    bool key = true;
    while (key) swap(&lock, &key);
    // critical section
    lock = false;
    // remainder section
} while (true);
```

- **Mutex (mutual exclusion) locks** (**spinlocks**):
  - Both `acquire()` and `release()` must be performed *atomically*.
  - A boolean variable `available` indicates if the lock is available or not.
  - Implementation of `acquire()` usually requires **busy waiting**, or **polling**, as seen in `test_and_set()` and `swap()`.
  - Spinlocks do have an advantage in that no time-consuming context switch is required when a process must wait on a lock.

```
void acquire() {
    while (!available);
    available = false;
} // an atomic operation
void release() {
    available = true;
} // an atomic operation
do {
    acquire(); // acquire lock
    // critical section
    release(); // release lock
    // remainder section
} while (true);
```

- **Semaphore**:
  - Synchronization tool that does not require busy waiting.
  - A semaphore `S` is an *integer* variable that, apart from initialization, is accessed only through two standard *atomic* operations: `wait()` and `signal()`.
  - **Counting semaphore**: an integer ranges over an unrestricted domain.
  - **Binary semaphore**: an integer ranges only between 0 and 1.

- *Binary* semaphores behave similarly to mutex locks. On systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.
- Each semaphore has an integer `value` and a list of processes or process control blocks (PCB) `list`.
- `value`: if negative, its magnitude is the number of processes waiting on that semaphore.
- `list`: when a process must wait on a semaphore, it is added to the list of processes.
- `block()`: when a process executes the `wait()` operation and finds that the semaphore value is not positive, it blocks itself and places itself into the waiting queue, and the state of the process is switched to the waiting state.
- `wakeup()`: a process that is blocked should be restarted and moved to the ready queue when some other process executes a `signal()` operation.

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
void wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add(S->list, self);
        block(self);
    }
} // an atomic operation
void signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove(S->list, other);
        wakeup(other);
    }
} // an atomic operation
do {
    wait(S);
    // critical section
    signal(S);
    // remainder section
} while (true);
```

- **Deadlock**: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- **Starvation** or **indefinite blocking** : processes wait indefinitely within the semaphore.
- **Priority inversion**: a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process.
- **Priority inheritance**: all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.

- Classical problems of synchronization: (1) bounded-buffer problem, (2) readers and writers problem, and (3) dining-philosophers problem.
- **Bounded-buffer problem**:
  - `mutex`: provides mutual exclusion for accesses to the buffer pool and is initialized to 1.
  - `empty`: counts the number of empty buffers and is initialized to n.
  - `full`: counts the number of full buffers and is initialized to 0.
  - Interpreted as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

```
do {
    // produce an item
    wait(empty);
    wait(mutex);
    // add the item to the buffer
    signal(mutex);
    signal(full);
} while (true);
```

```
do {
    wait(full);
    wait(mutex);
    // remove an item from buffer
    signal(mutex);
    signal(empty);
    // consume the item
} while (true);
```

- **Readers and writers problem**:
  - Allow multiple readers to read at the same time, but only one writer can write at any given time.
  - `mutex`: protects `readcount` from race condition and is initialized to 1.
  - `wrt`: is a write lock and initialized to 1.
  - `readcount`: counts the number of readers and is initialized to 0.

```
do {
    wait(wrt);
    // writing is performed
    signal(wrt);
} while (true);
```

```
do {
    wait(mutex);
    readcount++;
    if (readcount == 1) wait(wrt);
    signal(mutex);
    // reading is performed
    wait(mutex);
```

```
    readcount--;
    if (readcount == 0) signal(wrt);
    signal(mutex);
} while (true);
```

- **Dining philosophers problem**:
  - A *deadlock* occurs when all five philosophers become hungry at the same time and each grabs her left chopstick.
  - Allow at most four philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available.
  - Asymmetric solution: an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    // eat for a while
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    // think for a while
} while (true);
```

2017-04-06

- A counting semaphore can be implemented with binary semaphores. How?
  - `S1`: protects `S->value`. Initialized to 1.
  - `S2`: to block on the counting semaphore. Initialized to 0.
  - There is potential error about this implementation.

```
void wait(semaphore *S) {
    wait(S1);
    S->value--;
    if (S->value < 0) {
        signal(S1);
        wait(S2);
    } else
        signal(S1);
} // an atomic operation
void signal(semaphore *S) {
    wait(S1);
    S->value++;
    if (S->value <= 0)
        signal(S2);
    signal(S1);
} // an atomic operation
```

- **Critical region**:
  - Syntax: `region v when B do S;`
  - Using 2 delays to avoid checking the condition frequently.
- **Monitor**:
  - An ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor.
  - Only one process may be active within the monitor at a time.
  - **Condition variables** allow only 2 operations `signal()` and `wait()`.
- Consider a conditional variable `x`, when the `x.signal()` operation is invoked by a process P, there exists a suspended process Q
  - **Signal and wait**: P waits until Q leaves the monitor.
  - **Signal and continue**: Q waits until P leaves the monitor.
- Dining philosophers problems can also be solved by *monitor (condition variables)*.
- Implementing a monitor using semaphores:
  - `mutex`: protects the monitor and is initialized to 1.
  - `next`: the signaling processes can use `next` to suspend themselves. Initialized to 0.
  - `next_count`: counts the number of processes waiting on `next`.

```
wait(mutex);
// critical section
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

```
struct condition {
    void wait() {
        x_count++;
        if (next_count > 0)
            signal(next);
        else
            signal(mutex);
        wait(x_sem);
        x_count--;
    } // an atomic operation
    void signal() {
        if (x_count > 0) {
            next_count++;
            signal(x_sem);
            wait(next);
            next_count--;
        }
    } // an atomic operation
}
```