**Justin Ciocoi**

**Oct. 3, 2023**

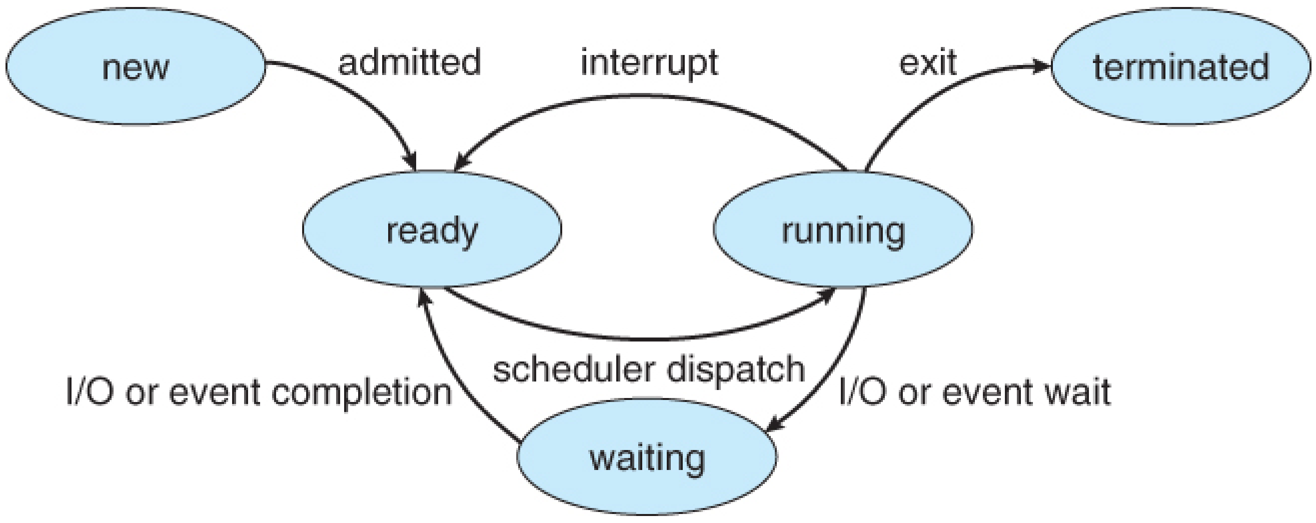
# CSCI 375 Textbook Notes

## Operating Systems

### Chapter 5: CPU Scheduling

##### 5.1: Basic Concepts

- The most basic idea of multiprogramming is to have some process running at all times such that CPU utilization is maximized
- CPU scheduling in order to divide CPU utilization is achieved through CPU scheduling
    - CPU scheduling is one of the **most fundamental** operating system functions
- **CPU-I/O Burst Cycle**
    - Process execution consists of a cycle between execution of instructions by the CPU and waiting for I/O operations
    - These "CPU Bursts" and "I/O Bursts" cycle between each other until the final CPU burst sends a system request to terminate execution
- **CPU Scheduler**
    - When the CPU does become idle, the CPU scheduler must choose one of the processes in the ready queue to be executed
    - The ready queue here is not necessarily a First-in First-out (FIFO) data structure, but can be implemented in a variety of ways
    - Below is a diagram representing the process state control mechanisms in an OS



- **Preemptive and Nonpreemptive Scheduling**
  - There are four conditions under which CPU scheduling decisions are made
    1. When a process switches from the running state to the waiting state
    2. When a process switches from the running state to the ready state
    3. When a process switches from the waiting state to the ready state
    4. When a process terminate
  - Under circumstances 1 and 4, there is no choice in terms of scheduling, and therefore we say that this scheduling scheme is nonpreemptive or *cooperative*
  - Otherwise, it is called *preemptive scheduling*
  - Preemptive scheduling can result in data race conditions leading to shared memory access inconsistencies
- **Dispatcher**
  - The dispatcher is a module involved in CPU scheduling which gives control of the CPU's core to the process selected by the scheduler
  - It involves the following processes
  - Switching contexts between processes
  - Switching to user mode
  - Jumping to the proper place in order to resume a program

- The dispatcher should prioritize efficiency since it will be invoked for every context switch
  - **Dispatch latency** is the time it takes for the dispatcher to stop one process and start another
- How often do context switches occur?
  - You can use the Linux terminal command `vmstat` to find the number of context switches
  - First line is average number of context switches per second since boot
  - Next two lines are the average number of switches in the last two 1-second periods

#### 5.2: Scheduling Criteria

- There are multiple different CPU scheduling algorithms and each can have different properties when it comes to selection of a new process
- The most substantial criteria when it comes to comparing CPU scheduling algorithms are as follows:
  - *CPU Utilization* which should be as close to 100% as possible
  - *Throughput* or the number of processes completed per time unit
  - *Turnaround time* or the amount of time it takes for a process to start and complete
  - *Waiting time* is the time spent by all processes in the waiting queue
  - *Response Time* often used instead of turnaround time as this is a more accurate measure of when a process is fully complete
- CPU Utilization and throughput should be maximized, whereas turnaround, waiting, and response time should be minimized

#### 5.3: Scheduling Algorithms

- **First-Come, First-Served Scheduling**
  - This is by far the *simplest* form of CPU scheduling algorithm
  - Whichever process requests the CPU first is allocated the CPU first

    - This is inefficient as easy to complete processes might be stuck behind particularly complex or long-lasting processes
    - This algorithm is *nonpreemptive*, so it poses many issues in any sort of interactive system
- **Shortest-Job-First Scheduling**
    - This scheduling algorithm looks at the CPU burst time of the next CPU burst for all processes
    - It finds the shortest next CPU burst, and when the CPU is next available, that process is started
    - When there is a tie between two shortest bursts, the first-come first-served algorithm is used to break the tie
    - There is no way for the CPU scheduler to know the approximate length of the next CPU burst, so the next CPU burst is predicted using an exponentially weighted trailing average
        - Similar to the prediction of round-trip-time that TCP does
- **Round-Robin Scheduling**
    - This scheduling algorithm is similar to the first-come first-served algorithm but also introduces preemption to allow the system to switch between processes
    - Uses a defined amount of time known as a *time quantum* or *time slice*
    - After each time slice, a context switch occurs
    - Thus, the smaller the time slice, the more efficient the Round-Robin algorithm can be since more context switches will occur per unit time given there are multiple processes ready to run
- **Priority Scheduling**
    - In this algorithm, the system allocates priority levels to different processes
    - These priority levels can be defined either internally or externally
    - *Starvation* is a major problem of priority scheduling, as processes ready to run might be blocked due to having to low of a priority level while higher priority processes execute

- **Multilevel Queue Scheduling**
    - This algorithm combines priority and round-robin scheduling
    - When multiple processes are at the same priority level, the round robin algorithm is implemented to select between them
- **Multilevel Feedback Queue Scheduling**
    - This is similar to the previous algorithm, but allows processes to switch queues such as moving from the background to the foreground

#### 5.4: Thread Scheduling

- On most modern operating systems, it is *kernel-level threads* - not processes - that are scheduled by the operating system
- *User-level threads* are managed by a user-level thread library of which the kernel is unaware
- In order to run on a CPU, user-level threads must be mapped to associated kernel-level threads through mapping that could be indirect or use a lightweight process, henceforth referred to as an LWP
- **Contention Scope**
    - Competition for the CPU takes place among different threads which belong to the same process
- **Pthread Scheduling**
    - Now we talk about the POSIX API that allows for the specification of PCS or SCS during thread creation
    - Pthreads identifies the following:
        - `PTHREAD_SCOPE_PROCESS` schedules threads using PCS scheduling
        - `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling
    - For systems that implement the many-to-many multithreading model:
        - `PTHREAD_SCOPE_PROCESS` schedules user-level threads onto available LWPs
        - `PTHREAD_SCOPE_SYSTEM` will create and bind an LWP for each user-level thread, which essentially serves to make the many-to-many machine operate on a one-to-one thread mapping model

        - The Pthread IPC provides functions for setting and getting the contention scope policy
            - `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
            - `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`
            - The first parameter for both functions contains a pointer to the attribute set of the thread
    - The following code illustrates a Pthread scheduling API

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char* argv[])
  {

      int i, scope;

      pthread_t tid[NUM_THREADS];
      pthread_attr_t attr;

      //get default attributes
      pthread_attr_init(&attr);

      //inquire on current scope
      if(pthread_attr_getscope(&attr, &scope)!=0)
          fprintf(stderr, "Unable to get scheduling scope\n");
      else
      {

          if(scope == PTHREAD_SCOPE_PROCESS)

              printf("PTHREAD_SCOPE_PROCESS");

          else if(scope == PTHREAD_SCOPE_SYSTEM)

              printf("PTHREAD_SCOPE_SYSTEM");
          else
              fprintf(stderr, "Illegal scope value.n\")
      }
      //now, set the scheduling algorithm either to PCS or SCS
      for(i=0;i<NUM_THREADS;i++)
          pthread_create(&tid[i], &attr, runner, NULL);

      //now join on each thread
      for(i=0;i<NUM_THREADS;i++)
          pthread_join(tid[i], NULL);
  }

  //Now, each thread will begin control in the following function
  void *runner(void *param)
  {

      //do something....

      //......
      //......
      pthread_exit(0);
  }
```

##### 5.5: Multi-Processor Scheduling

- So far, the discussion has centered mostly around systems that utilize only a single CPU core
- In a multi-core system, the issue of scheduling threads and processes becomes considerably more complex than in a single core system
- The term *multiprocessor* can refer to any of the following
    - Multicore CPU systems (most consumer systems)
    - Multithreaded Cores (most modern consumer systems)
    - NUMA systems
    - Heterogeneous multiprocessing
- **Approaches to multi-processor scheduling**
    - *Asymmetric* processing
        - Only one core access the system data structures
        - Introduces potential bottlenecks where system performance might be impacted
    - *Symmetric* processing
        - Each processor is self-scheduling
        - Each processor has a schedule that will examine the ready queue and select a thread to run based on the implemented scheduling algorithm
        - There are two possible thread organization structures when using symmetric multiprocessing
            - All threads can be stored in a common ready queue
            - Each processor can have its own private queue of threads
            - The first structure introduces potential data races and data dependencies, whereas the second structure is less efficient on average due to threads being limited to certain processors
- **Multicore Processors**

  - In modern CPUs, each core retains the architectural state of a traditional CPU and thus each core is recognized by the system as its own logical CPU
  - Because CPUs operate at a much higher speed than RAM, the CPU might wait significant amounts of time waiting for memory access to become available
    - This is known as a **memory stall**
    - It can also occur because of a cache miss
  - Multithreaded processing cores now contain two (or more) hardware threads such that if one hardware thread stalls while waiting for memory, another hardware thread can be used
  - The technique where a single physical CPU acts as many logical CPUs is known as chip multithreading (CMT)
  - Modern Intel processors use the term *hyper-threading* to describe this technique
    - The Intel i5-12400f, the CPU in my gaming PC, has 6 cores, with two threads per core, leading to 12 logical CPUs on the system
    - The Oracle Sparc M7 processor supports 8 threads per core, of which it has 8, so 64 logical CPUs
  - In general, there are two ways to multithread a processing core
    - *Coarse-grained* multithreading
      - A thread executes on a core until a long latency event (such as a memory stall)
      - Comes with a high cost of thread-switching
    - *Fine-grained* multithreading
      - Threads switch on a finer level of granularity, typically at the boundary of an instruction cycle
      - Thus, the cost of thread switching is considerably lower
- **Load Balancing**
  - In order to keep the utilization of a system near maximum, the balancing of loads between processing cores is an important topic

    - Load balancing is mostly dealt with in systems that use a private queue for each processing core, as a shared queue would make load balancing occur implicitly

- **Processor Affinity**
    - If a process must switch between threads, the original cache must be invalidated and the new cache must be repopulated such that the process can continue running
    - Because of the high overhead of switching CPU caches, most operating systems prefer to keep processes running on the same processor and take advantage of a *warm cache* which still contains all relevant data
    - This is known as **processor affinity**
        - This means that a process has an affinity for the processor on which it is currently running
    - If a system is using private ready queues for each thread, processor affinity is implicitly achieved as processes will not move between different processors
    - Multiple forms of processor affinity
        - *Soft affinity* where an operating system has a policy of attempting to keep a process running on the same processor
        - *Hard affinity* where an operating system has a policy of keeping a process running only on a specified subset of processors
- **Heterogeneous Multiprocessing**
    - In order to better manage power consumption and efficiency, heterogeneous multiprocessing allows different CPU cores tasked with similar instruction sets to operate at different clock speeds and power consumption levels
    - Think of Intel's *P-Cores* (Performance Cores) and *E-Cores* (Efficiency Cores)

#### 5.6: Real-Time CPU Scheduling

- **Minimizing Latency**
    - Two types of latency
        - *Interrupt Latency* which refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt (i.e. the "reaction time" of the CPU)

        - *Dispatch Latency* refers to the amount of time required for the scheduling dispatcher to stop one process and start another
    - In dispatch latency, the conflict phase has two components
        - The preemption of any process running in the kernel
        - A transfer of resources from low-priority processes to higher-priority processes
- **Priority-Based Scheduling**
    - In a *real-time* operating system, the most important feature is an immediate response to a real-time process as soon as that process requires the CPU
- **Rate-Monotonic Scheduling**
    - This scheduling algorithm schedules periodic tasks using a static priority policy with preemption
    - If a low priority process is running, and a higher priority process becomes available to run, the first process will be preempted
    - The shorter the period of a task, the higher its priority
    - Rate-Monotonic scheduling assumes that time of each CPU burst is *identical*
- **Earliest Deadline First Scheduling**
    - This scheduling algorithm assigns priority levels to processes dynamically according to their deadline
        - The earlier the deadline of a process, the higher priority it will be given
    - Under this system, whenever a process becomes runnable, it must make its *deadline requirements* known to the system
- **Proportional Share Scheduling**
    - In a proportional share scheduler, $T$ shares are allocated among all applications and an application can receive $N$ shares of time
        - Each application will thus have $\frac{N}{T}$ of the total processor time
    - These schedulers must work together with an admission-control policy in order to guarantee that an application receives its allocated shares of time

- **POSIX Real-Time Scheduling**
  - Here, the POSIX API for scheduling real-time threads is covered
  - POSIX defines two scheduling classes for real-time threads
    - `SCHED_FIFO` which schedules threads according to a first-come first-served policy
    - `SCHED_RR` which schedules threads according to a round-robin policy
    - `SCHED_OTHER` is also provided by POSIX, but its implementation is undefined and it may behave differently on different systems
    - Similar to getting and setting the scope, POSIX API provides functions for setting and getting the scheduling policy, namely
      - `pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy)`
      - `pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)`
  - The following is an implementation of POSIX real-time scheduling

```c
    #include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{

    int i, policy;

    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    //get default attributes
    pthread_attr_init(&attr);

    //get current scheduling policy
    if(pthread_attr_getschedpolicy(&attr, &policy)!=0)
        fprintf(stderr, "Unable to get policy\n");
    else
    {

        if(policy==SCHED_OTHER)

            printf("SCHED_OTHER\n");
        else if(policy==SCHED_RR)
            printf("SCHED_RR\n");
        else if(policy==SCHED_FIFO)
            printf("SCHED_FIFO\n");
    }

    //set the scheduling policy
    if(pthread_attr_getschedpolicy(&attr, SCHED_FIFO)!=0)
        fprintf(stderr, "Unable to set policy\n");

    //create threads
    for(i=0;i<NUM_THREADS;i++)
        pthread_create(&tid[i], NULL);

    //join on each thread
    for(i=0;i<NUM_THREADS;i++)
        pthread_join(tid[i], NULL);
}

//threads begin control in this function
void *runner(void *param)
{

    //do some work
    pthread_exit(0);
}
```

5.7: Operating System Examples

- **CPU Scheduling Examples in Linux**
    - Traditionally, Linux used the UNIX scheduling algorithm which was not designed with multiprocessing systems in mind
    - In Linux, scheduling is based on *scheduling classes* which are each assigned a different priority
        - Through the use of scheduling classes, Linux is able to implement a variety of different scheduling algorithms based on different situations
    - Rather than using strict priority values, Linux assigns a proportion of CPU processing time to each task
        - It does this using *nice values* which range from -20 to 19
        - It assigns a process a *targeted latency*, which is an interval of time during which every runnable task should run at least once
- **CPU Scheduling Examples in Windows**
    - The portion of the windows kernel that handles scheduling is called the *dispatcher*
    - Windows schedules use a priority-based, preemptive scheduling algorithm
    - The Windows API identifies the following six priority classes to which a process can belong
        - `IDLE_PRIORITY_CLASS`
        - `BELOW_NORMAL_PRIORITY_CLASS`
        - `NORMAL_PRIORITY_CLASS`
        - `ABOVE_NORMAL_PRIORITY_CLASS`
        - `HIGH_PRIORITY_CLASS`
        - `REALTIME_PRIORITY_CLASS`

- Typically, processes are members of the `NORMAL_PRIORITY_CLASS` unless its parent was a part of the `IDLE_PRIORITY_CLASS` or if another class was specified when the process was created
- In Windows, the priority class of a process, except `REALTIME_PROCESS_CLASS` is variable and can be changed using the `SetPriorityClass()` function in the Windows API
- Within a priority class, a process can have a relative priority from the following values
  - `IDLE`
  - `LOWEST`
  - `BELOW_NORMAL`
  - `NORMAL`
  - `ABOVE_NORMAL`
  - `HIGHEST`
  - `TIME_CRITICAL`
- Windows distinguishes between *foreground* and *background* processes and increases the scheduling quantum for foreground processes to assist user experience
- Windows 7 introduced *user-mode scheduling* which allows applications to create and manage threads completely independently of the kernel

#### 5.8: Algorithm Evaluation

- What are the most important criteria when it comes to selecting a CPU scheduling algorithm?
  - Maximizing CPU utilization
  - Maximizing throughput of a processor such that the average turnaround time is linearly proportional to the total execution time
- **Deterministic Modeling**
  - Deterministic modeling uses analytical evaluation using the given algorithm to produce some formula, equation, or number that describes or evaluates the performance of that algorithm
  - Deterministic modeling can be good for identifying trends across large data sets

- **Queuing Models**
  - Because the processes that will be run can be unpredictable on many end systems, no static set of processes can be used to do deterministic modeling
  - However, the distribution of CPU and I/O bursts can be measured and subsequently approximated
  - *Queuing network analysis* is the area of study that involves treating the CPU scheduling system as a network of connected devices
    - Let $n$ be the average long-term queue length
    - Let $\lambda$ be the average arrival rate for new processes in the queue
    - Let $W$ be the average waiting time in a queue
    - **Little's Formula** $n = \lambda * W$
- **Simulation**
  - Simulations of different loads on different scheduling algorithms can be used to test their efficacy across different scenarios
- **Implementation**
  - Even a simulation might not be perfectly suited for testing, so implementations must also be put in place and tested in order to adequately select the correct scheduling algorithm for any given task