Process Scheduling

The Operating System Kernel

- Basic set of primitive operations and processes
  - *Primitive*
    - Like a function call or macro expansion
    - Part of the calling process
    - Critical section for the process
  - *Process*
    - Synchronous execution with respect to the calling process
    - Can block itself or continuously poll for work
    - More complicated than primitives and more time and space
- Provides a way to provide protected system services, like *supervisor call instruction*
  - Protects the OS and key OS data structures (like process control blocks) from interference by user programs
  - The fact that a process is executing in kernel mode is indicated by a bit in program status word (PSW)
- Process classification
  1. Interactive processes
     - Communicate constantly with users; receive commands/data from keyboard/mouse and present results on screen or another device
     - Must allow a process to perform I/O and reasonable computation in a timely manner
     - Spend a lot of time waiting for keypresses and mouse operations
     - When there is input, must respond quickly , or user will find the system unresponsive
     - Typical delay beween 50-150ms
     - Variance of delay must be bounded or user will find the system to be erratic
  2. Batch processes
     - No user interaction
     - Run in the background
     - Need not be very responsive and hence, are penalized by scheduler
     - May be I/O-bound or CPU-bound
  3. Real-time processes
     - Stringent scheduling requirements
     - Should never be blocked by a lower priority process
     - Short guaranteed response time with minimum variance
- Execution of kernel
  - Nonprocess kernel
    - Kernel executes outside of any user process
    - Common on older operating systems
    - Kernel takes over upon interrupt or system call
    - Runs in its own memory, and has its own system stack
    - Concept of process applies only to user programs and not to OS
  - Execution with user processes
    - OS executes in the context of user process

    - OS is considered to be a collection of functions called by user processes to provide a service
    - Each user process must have memory for program, data, and stack areas for kernel routines
    - A separate *kernel stack* is used to manage code execution in kernel mode
    - OS code and data are in shared address space and are shared by all processes
    - Interrupt, trap, or system call execute in user address space but in kernel mode
    - Termination of kernel's job allows the process to run with just a mode switch back to user mode
  - Process-based kernel
    - Kernel is implemented as a collection of system processes, or microkernels
    - Modular OS design with a clean interface between different system processes

- Set of kernel operations
  - Process Management: Process creation, destruction, and interprocess communication; scheduling and dispatching; process switching; management of process control blocks
  - Resource Management: Memory (allocation of address space; swapping; page and segment management), secondary storage, I/O devices, and files
  - Input/Output: Transfer of data between memory and I/O devices; buffer management; allocation of I/O channels and devices to processes
  - Interrupt handling: Process termination, I/O completion, service requests, software errors, hardware malfunction

- Kernel in Unix
  - Controls the execution of processes by allowing their creation, termination, suspension, and communication
  - Schedules processes *fairly* for execution on CPU
    - CPU executes a process
    - Kernel suspends process when its time quantum elapses
    - Kernel schedules another process to execute
    - Kernel later reschedules the suspended process
  - Allocates main memory for an executing process
  - Allocates secondary memory for efficient storage and retrieval of user data
  - Allows controlled peripheral device access to processes

- Linux kernel
  - Monolithic, composed of several logically different components
  - Can dynamically load/unload some portions of kernel code, such as device drivers
    - Such code is made up of modules that can be automatically loaded or unloaded on demand
  - Lightweight processes (LWP) in Linux
    - Provided for better support for multithreaded applications
    - LWPs may share the same resources such as address space and open files
      - Modification of a shared resource by one LWP is immediately visible to the other
      - Processes need to synchronize before accessing shared resource
    - Each thread/LWP can be scheduled independently by the kernel
    - Thread group
      - Set of LWPs to implement a multithreaded application
      - Act as a whole for some system calls such as `getpid()`, `kill()` and `exit()`
  - Uses kernel threads in a very limited way
    - Kernel thread is an execution context that can be independently scheduled
      - May be associated with a user program or may run only some kernel functions

        * Context switch between kernel threads is much less expensive than context switch between ordinary processes
            · Kernel threads usually operate on a common address space
        * Linux uses kernel threads to execute a few kernel functions periodically; they do not represent the basic execution context abstraction
    - Defines its own version of lightweight processes
        * Different from SVR4 and Solaris that are based on kernel threads
        * Linux regards lightweight processes as basic execution context and handles them via the nonstandard `clone()` system call
    - Preemptive kernel
        * May be compiled with "Preemptible kernel" option (starting with Linux 2.6)
        * Can arbitrarily interleave execution flows while they are in privileged mode
    - Multiprocess support
        * Linux 2.6 onwards supports SMP for different memory models, including Non-Uniform Memory Acces (NUMA)
        * A few parts of the kernel code are still serialized by means of a single *big kernel lock*

- Highest Level of User Processes: The *shell* in Unix
    - Created for each user (login request)
    - Initiates, monitors, and controls the progress for user
    - Maintains global accounting and resource data structures for the user
    - Keeps static information about user, like identification, time requirements, I/O requirements, priority, type of processes, resource needs
    - May create child processes (progenies)

- Process image
    - Collection of programs, data, stack, and attributes that form the process
    - User data
        * Modifiable part of the user space
        * Program data, user stack area, and modifiable code
    - User program
        * Executable code
    - System stack
        * Used to store parameters and calling addresses for procedure and system calls
    - Process control block
        * Data needed by the OS to control the process
    - Location and attributes of the process
        * Memory management aspects: contiguous or fragmented allocation

Synchronization and critical regions

- Needed to avoid race conditions among non-reentrant portion of kernel code
- Protect the modification and observation of resource counts
- Synchronization achieved by
    - Disabling kernel preemption
        * A process executing in kernel mode may be made nonpreemptive

    - If a process in kernel mode voluntarily relinquishes the CPU it must make sure that all data structures are left in consistent state; upon resumption, it must recheck the values of any previously accessed data structures
    - Works for uniprocessor machines but not on a machine with multiple CPUs
  - Disable interrupts
    - If the critical section is large, interrupts remain disabled for a relatively long time, potentially causing all hardware activities to freeze
    - Does not work on multiprocessor system
  - Spin locks
    - Implementation of busy-wait in some systems, including Linux
    - Convenient in the kernel code because kernel resources are locked just for a fraction of a millisecond
      - More time consuming to release the CPU and reacquire it later
      - If the time required to update data structure is short, semaphore could be very inefficient
      - Process checks semaphore and suspends itself (expensive operation); other process meanwhile may have released the semaphore
    - Data structures need to be protected from being concurrently accessed by kernel control paths that run on different CPUs
    - When a process finds the lock closed by another process, it spins around repeatedly till the lock is open
    - Spin lock useless in uniprocessor environment
  - Semaphores
    - Implement a locking primitive that allows a waiting process to sleep until the desired resource is available
    - Two types of semaphores in Linux: kernel semaphores and IPC semaphores
    - Kernel semaphores
      - Similar to spin locks
      - When a kernel control path tries to acquire a busy resource protected by kernel semaphore, it is suspended; it becomes runnable again when the resource is released
      - Kernel semaphores can be acquired only by functions that are allowed to sleep (no interrupt handlers or deferable functions)

```
struct semaphore
{
    atomic_t  count;   // Number of available resources
       // If < 0, no resource available and at least one
       // process waiting for resource
    queue_t   wait;    // Sleeping processes that are waiting for resource
       // If count > 0, this queue is empty
    int       sleepers; // Processes sleeping on semaphore?
};
```

  - Avoiding deadlocks
    - An issue when the number of kernel locks used is high
    - Difficult to ensure that no deadlock state will ever be reached for all possible ways to interleave kernel control paths
    - Linux avoids this problem by requesting locks in a predefined order

Data Structures for Processes and Resources

- Used by the OS to keep track of each process and resource
- Cross-referenced or linked in main memory for proper coordination
- Memory tables

  - Used to keep track of allocated and requested main and secondary memory
  - Protection attributes of blocks of main and secondary memory
  - Information to map main memory to secondary memory
- I/O tables
  - Used to manage I/O devices and channels
  - State of I/O operation and location in main memory as source/destination of operation
- File tables
  - Information on file existence, location in secondary memory, current status, and other attributes
  - Part of file management system
- Process control block
  - Most important data structure in an OS
  - Set of all process control blocks describes the *state* of the OS
  - Read and modified by almost every subsystem in the OS, including scheduler, resource allocator, and performance monitor
  - Constructed at process creation time
    * Physical manifestation of the process
    * Set of data locations for local and global variables and any defined constants
  - Contains specific information associated with a specific process
    * The information can be broadly classified as process identification, processor state information, and process control information
    * Can be described by Figure **??**
    * *Identification.* Provided by a pointer to the PCB
      · Always a unique integer in Unix, providing an index into the primary process table
      · Used by all subsystems in the OS to determine information about a process
      · Used for cross-referencing in other tables (memory, I/O)
    * *CPU state*
      · Provides snapshot of the process
      · The program counter register is initialized to the entry point of the program (`main()` in C programs)
      · While the process is running, the user registers contain a certain value that is to be saved if the process is interrupted
      · Typically described by the registers that form the processor status word (PSW) or *state vector*
      · Exemplified by EFLAGS register on Pentium that is used by any OS running on Pentium, including Unix and Windows NT
    * *Process State.* Current activity of the process
      · Running. Executing instructions.
      · Ready. Waiting to be assigned to a processor; this is the state assigned when the data structure is constructed
      · Blocked. Waiting for some event to occur.
    * Allocated address space (Memory map)
    * Resources associated with the process (open files, I/O devices)
    * *Other Information.*
      · Progenies/offsprings of the process
      · Priority of the process
      · Accounting information. Amount of CPU and real time used, time limits, account numbers, etc.

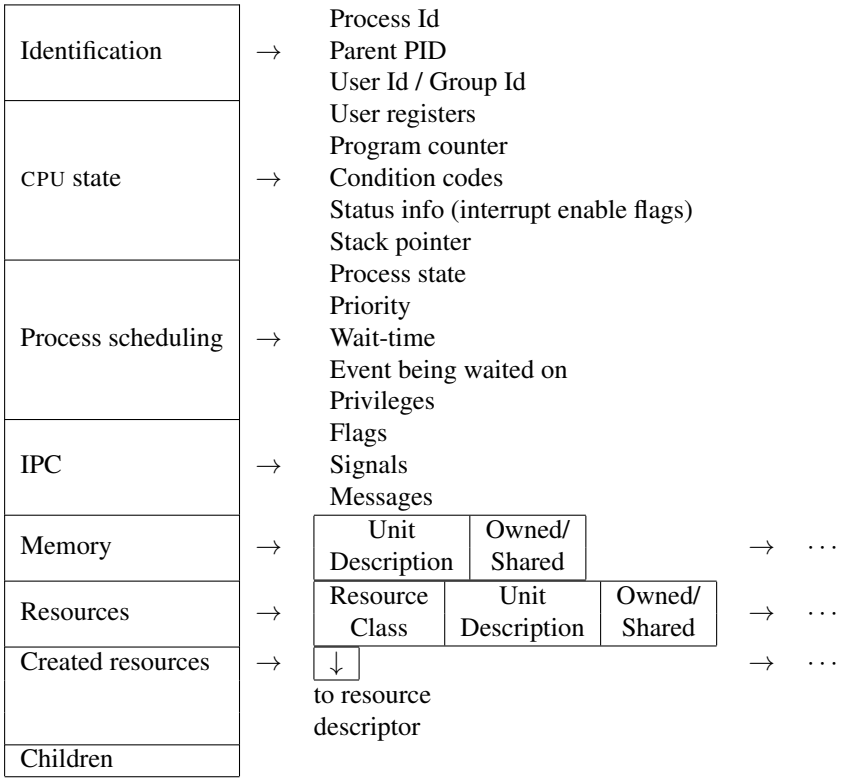
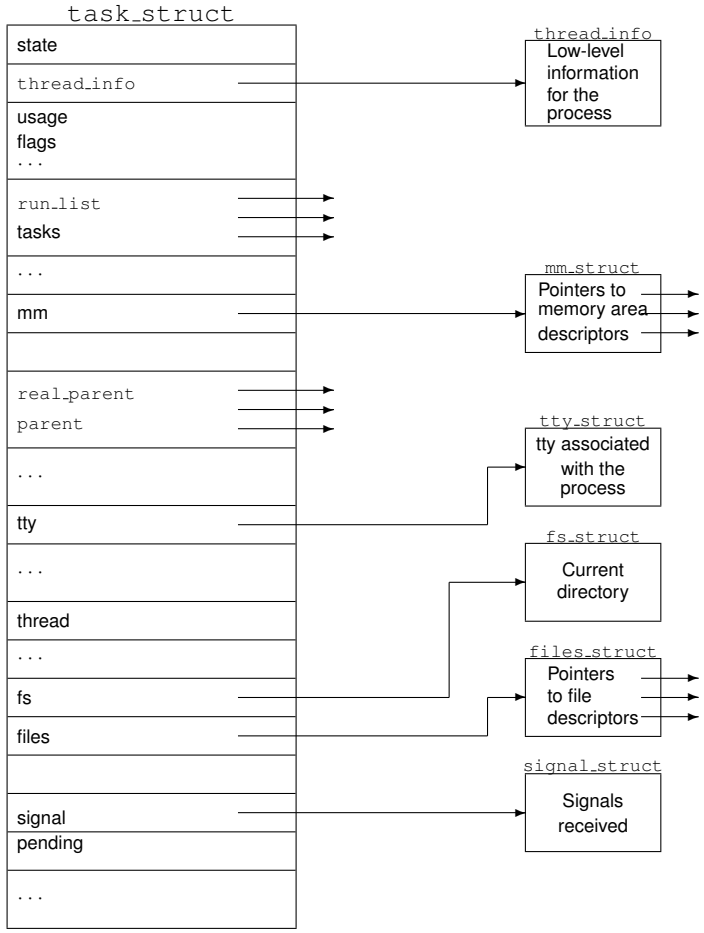


Figure 1: Process Control Block

- I/O status information. Outstanding I/O requests, I/O devices allocated, list of open files, etc.

– Linux task structure



- Process identification in Linux
  - Linux identifies a process by 32-bit address of `task_struct`, called a *process descriptor pointer*
  - Unix identifies a process by process id (or PID) stored in the `pid` field of process descriptor
    - PIDs are numbered sequentially; they are assigned in an increasing order and are reused in the long run
    - Process IDs are not reused immediately to prevent race conditions
      - A process sends a signal to another process
      - Before the signal is received, the recipient has terminated and the process ID reassigned to another newly created process
      - The signal is sent to a wrong process
    - Maximum PID number is 32,767 (`PID_MAX_DEFAULT - 1`)
    - Default can be increased or decreased by writing to the file `/proc/sys/kernel/pid_max`
    - In 64-bit architectures, the maximum PID number can be changed to 4,194,303
    - PIDs managed by a bitmap `pidmap_array`
    - Since a page frame contains 32,768 bits (4KB), the 32-bit architectures allow the storage of `pidmap_array` bitmap into a single page
    - In 64-bit architectures, additional pages are added to bitmap when kernel assigns a PID number too large for the current bitmap
  - POSIX standard requires the threads in the same group to have a common PID to send a signal to affect all threads in the group
    - The PID of the *thread group leader* (first LWP in the group) is shared by all threads
    - Stored in the `tgid` field of process descriptor

  - Linux stores process information in process descriptor pointers linked by a doubly linked list
    * The head of the list is process 0 or `swapper`, identified by `task_struct init_task`
    * `tasks->prev` field of `init_task` points to the `tasks` field of process descriptor inserted last in the list
    * The entire process list is scanned by the following macro:

```
#define for_each_process(p) \
    for ( p = &init_task; \
          ( p = list_entry((p)->tasks.next, struct task_struct, tasks ) ) \
              != &init_task; )
```

  - Lists of `TASK_RUNNING` processes
    * When looking for a new process to run on a CPU, kernel has to consider only the runnable processes (processes in the `TASK_RUNNING` state)
    * Earlier Linux versions put all runnable processes in the same list called `runqueue`
  - Scheduler speedup is achieved by splitting `runqueue` into many lists of runnable processes, one list per priority
    * Linux assigns a priority in the range of 0 to 139
    * 140 different lists in `runqueue` structure
    * Process descriptor linked into the list of runnable processes with the same priority
    * On multiprocessor systems, each CPU has its own `runqueue`
    * Implemented through the data structure `prio_array_t`

```
struct prio_array_t
{
    int               nr_active;   // Number of active processes
    unsigned long     bitmap[5];   // Priority bitmap; set if the corresponding
                                   // priority list is not empty
    struct list_head queue[140];  // Head of each priority list
}
```

    * Inserting a processes `p` into the runqueue

```
list_add_tail ( &p->run_list, &array->queue[p->prio] );
__set_bit ( p->prio, array->bitmap );
array->nr_active++;
p->array = array;
```

- Thread control block
  - Each thread provides an independent execution activity within a process
  - Replicates only the bare minimum information needed to schedule and manage thread execution
  - CPU state, PC, stack pointer; process state replaced by thread state
  - Code, global data, resources, and open files are shared among all threads

- Resource Descriptors.
  - Resource.
    * Reusable, relatively stable, and often scarce commodity
    * Successively requested, used, and released by processes
    * Hardware (physical) or software (logical) components
  - Resource class.
    * *Inventory.* Number and identification of available units
    * *Waiting list.* Blocked processes with unsatisfied requests
    * *Allocator.* Selection criterion for honoring the requests
  - Contains specific information associated with a certain resource

    - `p^.status_data` points to the waiting list associated with the resource.
    - Dynamic and static resource descriptors, with static descriptors being more common
    - Identification, type, and origin
        - Resource class identified by a pointer to its descriptor
        - Descriptor pointer maintained by the creator in the process control block
        - External resource name ⇒ unique resource id
        - Serially reusable or consumable?
    - Inventory List
        - Associated with each resource class
        - Shows the availability of resources of that class
        - Description of the units in the class
    - Waiting Process List
        - List of processes blocked on the resource class
        - Details and size of the resource request
        - Details of resources allocated to the process
    - Allocator
        - Matches available resources to the requests from blocked processes
        - Uses the inventory list and the waiting process list
    - Additional Information
        - Measurement of resource demand and allocation
        - Total current allocation and availability of elements of the resource class

- Context of a process
    - Information to be saved that may be altered when an interrupt is serviced by the interrupt handler
    - PC and stack pointer registers
    - General purpose registers
    - Floating point registers
    - Processor control registers (PSW) containing information on CPU state
    - Memory management registers used to keep track of RAM accessed by process
    - Process switch/Task switch/Context switch
        - Suspend the currently running process and resume the execution of some process previously suspended
        - Hardware context
            - All processes share the CPU registers
            - Before resuming a process, kernel must make sure that each register is loaded with the value it had when the process was suspended
            - Subset of the process execution context
            - Part of hardware context is stored in process descriptor while remaining part is stored in kernel mode stack
    - Difference between context switch and interrupt handling
        - Code executed by interrupt handler does not constitute a process
        - Just a kernel control path that runs at the expense of the same process that was running when the interrupt occurred

- Process groups
    - Represent a *job* abstraction
    - As an example, processes in a pipeline form a group and the shell acts on those as a single entity
    - Process descriptor contains a field called *process group* ID

    - PID of the *group leader*
  - Login session
    - All processes that are descendants of the process that started a working session on a specific terminal
    - All processes in a process group are in the same login session
    - A login session may have several process groups
    - One of the processes is always in the foreground; it has access to the terminal
    - When a background process tries to access the terminal, it receives a `SIGTTIN` or `SIGTTOUT` signal

Basic Operations on Processes and Resources

- Implemented by kernel primitives
- Maintain the *state* of the operating system
- Indivisible primitives protected by "busy-wait" type of locks
- *Process Control Primitives*
  - `create` – Establish a new process
    - Assign a new unique process identifier (PID) to the new process
    - Allocate memory to the process for all elements of process image, including private user address space and stack; the values can possibly come from the parent process; set up any linkages, and then, allocate space for process control block
    - Create a new process control block corresponding to PID and add it to the process table; initialize different values in there such as parent PID, list of children (initialized to `null`), program counter (set to program entry point), system stack pointer (set to define the process stack boundaries)
    - Initial CPU state, typically initialized to *Ready* or *Ready, suspend*
    - Add the process id of new process to the list of children of the creating (parent) process
    - Assign initial priority
      - Initial priority of the process may be greater than the parent's
    - Accounting information and limits
    - Add the process to the *ready list*
  - `suspend`. Change process state to suspended
    - A process may suspend only its descendants
    - May include cascaded suspension
    - Stop the process if the process is in *running state* and save the state of the processor in the process control block
    - If process is already in *blocked state*, then leave it blocked, else change its state to *ready state*
    - If need be, call the `scheduler` to schedule the processor to some other process
  - `activate`. Change process state to active
    - Change one of the descendant processes to *ready state*
    - Add the process to the *ready list*
  - `destroy`. Remove one or more processes
    - Cascaded destruction
    - Only descendant processes may be destroyed
    - If the process to be "killed" is running, stop its execution
    - Free all the resources currently allocated to the process
    - Remove the process control block associated with the killed process
  - `change_priority`. Set a new priority for the process

  - Change the priority in the process control block
  - Move the process to a different queue to reflect the new priority

- Resource Primitives
  - `create_resource_class`. Create the descriptor for a new resource class
    - Dynamically establish the descriptor for a new resource class
    - Initialize and define inventory and waiting lists
    - Criterion for allocation of the resources
    - Specification for insertion and removal of resources
  - `destroy_resource_class`. Destroy the descriptor for a resource class
    - Dynamically remove the descriptor for an existing resource class
    - Resource class can only be destroyed by its creator or an ancestor of the creator
    - If any processes are waiting for the resource, their state is changed to *ready*
  - `request`. Request some units of a resource class
    - Includes the details of request – number of resources, absolute minimum required, urgency of request
    - Request details and calling process-id are added to the waiting queue
    - Allocation details are returned to the calling process
    - If the request cannot be immediately satisfied, the process is blocked
    - Allocator gives the resources to waiting processes and modifies the allocation details for the process and its inventory
    - Allocator also modifies the resource ownership in the process control block of the process
  - `release`. Release some units of a resource class
    - Return unwanted and serially reusable resources to the resource inventory
    - Inform the allocator about the return

Organization of Process Schedulers

- Objective of Multitasking: Maximize CPU utilization and increase *throughput*
- Two processes P_0 and P_1

P_0	t_0	i_0	t_1	i_1	$\cdots$	i_{n-1}	t_n

P_1	t'_0	i'_0	t'_1	i'_1	$\cdots$	i'_{m-1}	t'_m

- Two processes P_0 and P_1 without multitasking

P_0	t_0	i_0	t_1	i_1	$\cdots$	i_{n-1}	t_n	P_0 terminated						
P_1	P_1 waiting							t'_0	i'_0	t'_1	i'_1	$\cdots$	i'_{m-1}	t'_m

- Processes P_0 and P_1 with multitasking

P_0	t_0		t_1		$\cdots$	t_n	
P_1		t'_0		t'_1	$\cdots$		t'_m

- Each entering process goes into *job queue*. Processes in job queue
  - reside on mass storage
  - await allocation of main memory

- Processes residing in main memory and awaiting CPU time are kept in *ready queue*
- Processes waiting for allocation of a certain I/O device reside in *device queue*
- *Scheduler*
  - Concerned with deciding a policy about which process to be dispatched
  - After selection, loads the process state or dispatches
  - Process selection based on a scheduling algorithm
- Autonomous *vs* shared scheduling
  - Shared scheduling
    - Scheduler is invoked by a function call as a side effect of a kernel operation
    - Kernel and scheduler are potentially contained in the address space of all processes and execute as a part of the process
  - Autonomous scheduling
    - Scheduler (and possibly kernel) are centralized
    - Scheduler is considered a separate process running autonomously
    - Continuously polls the system for work, or can be driven by wakeup signals
    - Preferable in multiprocessor systems as master/slave configuration
      - One CPU can be permanently dedicated to scheduling, kernel and other supervisory activities such as I/O and program loading
      - OS is clearly separated from user processes
    - *Who dispatches the scheduler?*
      - Solved by transferring control to the scheduler whenever a process is blocked or is awakened
      - Scheduler treated to be at a higher level than any other process
  - Unix scheduler
    - Autonomous scheduler
    - Runs between any two other processes
    - Serves as a dummy process that runs when no other process is ready
- Short-term v/s Long-term schedulers
  - Long-term scheduler
    - Selects processes from job queue
    - Loads the selected processes into memory for execution
    - Updates the ready queue
    - Controls the *degree of multiprogramming* (the number of processes in the main memory)
    - Not executed as frequently as the short-term scheduler
    - Should generate a good mix of CPU-bound and I/O-bound processes
    - May not be present in some systems (like time sharing systems)
  - Short-term scheduler
    - Selects processes from ready queue
    - Allocates CPU to the selected process
    - Dispatches the process
    - Executed frequently (every few milliseconds, like 10 msec)
    - Must make a decision quickly $\Rightarrow$ must be extremely fast

Process or CPU Scheduling

- Major task of any operating system – allocate ready processes to available processors
  - *Scheduler* decides the process to run first by using a *scheduling algorithm*
  - Effectively a matter of managing queues to minimize queueing delay and to optimize performance in a queueing environment
  - Two components of a scheduler
    1. Process scheduling
       * Decision making policies to determine the order in which active processes compete for the use of CPU
    2. Process dispatch
       * Actual binding of selected process to the CPU
       * Involves removing the process from ready queue, change its status, and load the processor state
- Desirable features of a scheduling algorithm
  - Fairness: Make sure each process gets its fair share of the CPU; also, no process should be starved for CPU
  - Efficiency: Keep the CPU busy 100% of the time
  - Response time: Minimize response time for interactive users
  - Turnaround: Minimize the time batch users must wait for output
  - Throughput: Maximize the number of jobs processed per hour
- Types of scheduling
  - Preemptive
    * Temporarily suspend the logically runnable processes
    * More expensive in terms of CPU time (to save the processor state)
    * Can be caused by

      Interrupt. Not dependent on the execution of current instruction but a reaction to an external asynchronous event

      Trap. Happens as a result of execution of the current instruction; used for handling error or exceptional condition

      Supervisor call. Explicit request to perform some function by the kernel
  - Nonpreemptive
    * Run a process to completion
- The Universal Scheduler: specified in terms of the following concepts
  1. Decision Mode
     - Select the process to be assigned to the CPU
  2. Priority function
     - Applied to all processes in the ready queue to determine the *current* priority
  3. Arbitration rule
     - Applied to select a process in case two processes are found with the same current priority

The Decision Mode

- Time (decision epoch) to select a process for execution
- Preemptive and nonpreemptive decision
- Selection of a process occurs
  1. when a new process arrives
  2. when an existing process terminates
  3. when a waiting process changes state to ready

4. when a running process changes state to waiting (I/O request)
5. when a running process changes state to ready (interrupt)
6. every q seconds (quantum-oriented)
7. when priority of a ready process exceeds the priority of a running process

– Selective preemption: Uses a bit pair (u_p, v_p)
u_p set if p may preempt another process
v_p set if p may be preempted by another process

The Priority Function

– Defines the priority of a ready process using some parameters associated with the process
– Memory requirements – Important due to swapping overhead
Smaller memory size ⇒ Less swapping overhead
Smaller memory size ⇒ More processes can be serviced
– Attained service time
Total time when the process is in the running state
– Real time in system
Total actual time the process spends in the system since its arrival
– Total service time
Total CPU time consumed by the process during its lifetime
Equals attained service time when the process terminates
Higher priority for shorter processes
Preferential treatment of shorter processes reduces the average time a process spends in the system
– External priorities
Differentiate between classes of user and system processes
Interactive processes ⇒ Higher priority
Batch processes ⇒ Lower priority
Accounting for the resource utilization
– Timeliness – Dependent upon the urgency of a task and deadlines
– System load
Maintain good response time during heavy load
Reduce swapping overhead by larger quanta of time

The Arbitration Rule

– Random choice
– Round robin (cyclic ordering)
– Chronological ordering (FIFO)

- Time-Based Scheduling Algorithms
  – May be independent of required service time
- First-in/First-out (FIFO) Scheduling
  – Also called First-Come-First-Served (FCFS), or strict queuing
  – Simplest CPU-scheduling algorithm
  – Nonpreemptive decision mode
  – Upon process creation, link its PCB to rear of the FIFO queue
  – Scheduler allocates the CPU to the process at the front of the FIFO queue
  – Average waiting time can be long

Process	Burst time
P_1	24
P_2	3
P_3	3

  - Let the processes arrive in the following order:

    $$P_1, P_2, P_3$$

    Then, the average waiting time is calculated from:

P_1		P_2		P_3	
1	24	25	27	28	30

  - Average waiting time = $\frac{0+24+27}{3} = 17$ units
  - Tends to favor CPU-bound processes compared to I/O-bound processes, because of its non-preemptive nature
    * Against cycle stealing
    * If a CPU-bound process is in wait state when the I/O-bound process initiates I/O, CPU goes idle
      · Inefficient use of both CPU and I/O devices

- Last-in/First-out (LIFO) Scheduling
  - Similar to FIFO scheduling
  - Average waiting time is calculated from:

P_3		P_2		P_1	
1	3	4	6	7	30

  - Average waiting time = $\frac{0+3+6}{3} = 3$ units
    * Substantial saving but what if the order of arrival is reversed.

- Shortest Job Next (SJN) Scheduling
  - Also called Shortest Job First (SJF) scheduling
  - Associate the length of the next CPU burst with each process
  - Assign the process with shortest CPU burst requirement to the CPU
  - Nonpreemptive scheduling
  - Specially suitable to batch processing (long term scheduling)
  - Ties broken by FIFO scheduling
  - Consider the following set of processes

Process	Burst time
P_1	6
P_2	8
P_3	7
P_4	3

    Scheduling is done as:

P_4		P_1		P_3		P_2	
1	3	4	9	10	16	17	24

    Average waiting time = $\frac{3+16+9+0}{4} = 7$ units
  - Using FIFO scheduling, the average waiting time is given by $\frac{0+6+14+21}{4} = 10.25$ units
  - Provably optimal scheduling – Least average waiting time
    * Moving a short job before a long one decreases the waiting time for short job more than it increase the waiting time for the longer process
  - Problem: To determine the length of the CPU burst for the jobs

- Shortest Remaining Time First (SRTF) Scheduling
  - Preemptive version of shortest job next scheduling

- Preemptive in nature (only at arrival time)
- Highest priority to process that need least time to complete
- Consider the following processes

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

- Schedule for execution

P_1	P_2	P_4	P_1	P_3
1 2	5 6	10 11	17 18	26

- Average waiting time calculations

- Round-Robin Scheduling
  - Preemptive in nature, based on fixed time slices or time quanta q
    - Reduces the penalty that short jobs suffer with FIFO
  - Generate clock interrupts at periodic intervals, or time slices or time quanta
    - Time quantum between 10 and 100 milliseconds
    - Each process gets a slice of time before being preempted
  - All user processes treated to be at the same priority
  - Ready queue treated as a circular queue, effectively giving a variation on FIFO
    - New processes added to the rear of the ready queue
    - Preempted processes added to the rear of the ready queue
    - Scheduler picks up a process from the head of the queue and dispatches it with a timer interrupt set after the time quantum
  - CPU burst $< q \Rightarrow$ process releases CPU voluntarily
  - Timer interrupt results in context switch and the process is put at the rear of the ready queue
  - No process is allocated CPU for more than 1 quantum in a row
  - Consider the following processes

Process	Burst time
P_1	24
P_2	3
P_3	3

  - $q = 4$ ms

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1
1 4	5 7	8 10	11 14	15 18	19 22	23 26	27 30

  - Average waiting time = $\frac{6+4+7}{3} = 5.66$ milliseconds
  - If there are n processes in the ready queue, and q is the time quantum, then each process gets $\frac{1}{n}$ of CPU time in chunks of at most q time units
    Hence, each process must wait no longer than $(n-1) \times q$ time units for its next quantum
  - Performance depends heavily on the size of time quantum
    - Large time quantum $\Rightarrow$ FIFO scheduling
    - Small time quantum $\Rightarrow$ Large context switching overhead
    - Rule of thumb: 80% of the CPU bursts should be shorter than the time quantum

- Multilevel Feedback Queue Scheduling
  - Most general CPU scheduling algorithm
  - Prefers shorter jobs by penalizing jobs that use too much CPU time, using a dynamically computed priority
  - Background
    - Make a distinction between foreground (interactive) and background (batch) processes
    - Different response time requirements for the two types of processes and hence, different scheduling needs
    - Separate queue for different types of processes, with the process priority being defined by the queue
  - Separate processes with different CPU burst requirements
  - Too much CPU time $\Rightarrow$ lower priority
  - I/O-bound and interactive process $\Rightarrow$ higher priority
  - n different priority levels – $\Pi_1 \cdots \Pi_n$
  - Each process may not receive more than T_Π time units at priority level Π
  - Let $T_\Pi = 2^{n-\Pi} T_n$, where T_n is the maximum time on the highest priority queue at level n
    - Each process will spend time T_n at priority n, and time $2^i T_n$ at priority $n - i$, $1 \leq i \leq n$
    - When a process receives time T_Π, decrease its priority to $\Pi - 1$
    - Process may remain at the lowest priority level for infinite time
  - CPU always serves the highest priority queue that has processes in it ready for execution
  - Variation: *Aging* to prevent starvation
- Performance of interactive scheduling algorithms
  - Response time
    - Time elapsed between pressing a key and some reaction by the system
    - Depends on the type of process (editor vs grammer checker)
  - Many processes in the system result in an increase in response time
    - Dependent on quantum and time for context switch
- Policy Driven CPU Scheduling
  - Based on a *policy function*
  - Policy function gives the correlation between actual and desired utilization for services
  - Services can be measured in terms of time units during which a particular resource is used by the process
  - Attempt to strike a balance between actual and desired resource utilization
- Real-time scheduling
  - Requires a small scheduling latency for the kernel
  - Unix processes cannot be preempted when they are running in kernel mode; this makes Unix unsuitable for real-time processing
  - Steady input to be processed before the next instance is received
  - Stringent scheduling requirements with deadlines for processing
  - Period or time interval in ms or μs to process each input item
    - If the process completes its task, it just stays idle till next period
    - End of period is the deadline that is the time by which the task needs to be completed
    - Video processor
      - Receives a frame every 16 ms
      - Period is 16 ms

        - If frame is processed in 10ms, it waits for 6ms for next frame
        - If frame is processed in 20ms, it can stop processing at 16ms and starts working on the next frame, or finishes processing the current frame at 20ms and drops the next frame
    - Should never be blocked by a lower priority process
    - Short guaranteed response time with minimum variance
    - Rate monotonic (RM) scheduling algorithm
        - Priority inversely proportional to period
        - Two processes p_0 and p_1 with period 4 and 5, respectively, and total CPU burst per period of 1 and 3, respectively
            - p_0 gets scheduled at times 0, 4, 8, 12, ...
            - p_1 gets scheduled at times 0, 5, 10, 15, ...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p_0	p_1	p_1	p_1	p_0	p_1	p_1	p_1	p_0		p_1	p_1	p_0	p_1		p_1	p_0	p_1	p_1

    - Earliest deadline first (EDF) scheduling
        - Schedule process with the shortest remaining time until the deadline
        - Consider the two processes from RM scheduling

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p_0	p_1	p_1	p_1	p_0	p_1	p_1	p_1	p_0		p_1	p_1	p_1	p_0		p_1	p_0	p_1	p_1

            - At time 16, p_0 and p_1 are both available with a deadline of 20
            - Let p_0 preempt p_1 since p_0 has higher priority due to shorter period (arbitration rule)
    - Performance of algorithms
        - A schedule is *feasible* if the deadlines of all processes can be met
        - If T_i is the total CPU time for process i and D_i is its period, the fraction of CPU time used by process i is given by $u_i = T_i/D_i$
        - CPU utlization U is given by the sum of individual fractions used by each process

$$U = \sum_{i=1}^{n} \frac{T_i}{D_i}$$

            - $U = 1$: CPU utilization is 100%
            - $U < 1$: Schedule is feasible
            - $U > 1$: No feasible schedule exists
        - EDF is an optimal algorithm; a feasible schedule is guaranteed if $U \leq 1$
        - RM does not guarantee a feasible schedule
            - Empirically, RM likely to produce a feasible schedule if $U < 0.7$
        - Two processes p_0 (period 8, CPU 3) and p_1 (period 10, CPU 6)

- Combination of real-time and regular processes
    - 2-tier approach
        - Schedule real-time processes at highest priority using FIFO because of short CPU bursts
        - Longer real-time processes can be scheduled at high priority using round-robin to evenly distribute the CPU time among processes
        - Schedule batch and interactive processes using multilevel feedback queue
    - 2-tier approach with floating priorities
        - Real-time processes with FIFO or round-robin using multi-level priorities
        - Interactive and batch processes with a variation of multi-level feedback

Scheduling policy in Linux

- Objectives include
  - Fast process response time
  - Good throughput for background jobs
  - Avoidance of process starvation
  - Reconcile the needs of low- and high-priority processes
- Based on time sharing – several processes run in time-multiplexing mode
- Process priority is dynamic
  - Scheduler keeps track of what the processes are doing and adjusts their priorities
  - Processes that do not get CPU for a long time will have their priority boosted by dyniamically increasing the priority
  - Processes that run for a long time are penalized by decreasing their priority
- Linux scheduling pre-2.6
  - At every process switch, kernel scanned the list of ready processes, computed their priorities, and selected the one with highest priority
  - Expensive algorithm especially if the list of ready processes is large
- Linux scheduling post-2.6
  - Designed to scale well with the number of ready processes
  - Selects the process to run in constant time, independent of the number of process in the ready queue
  - Designed to scale well with the number of processors
    - Each CPU has its own `runqueue`
  - Does a better job of distinguishing between interactive and batch processes
  - Scheduler always finds a process to be executed
    - There is always at least one runnable process
      - `swapper` with PID 0
      - Executes only when CPU cannot execute any other process
      - Every CPU in a multiprocessor system has its own `swapper` with PID 0
  - Every Linux process scheduled according to one of *scheduling classes*

    `SCHED_FIFO` * A FIFO real-time process
      - When a scheduler assigns CPU to the process, it leaves the process descriptor in its current position in `runqueue`
      - If there is no other higher priority runnable real-time process, the current process uses CPU as long as it needs even if there are other runnable real-time processes at the same priority

    `SCHED_RR` * A round robin real-time process
      - When a scheduler assigns CPU to the process, it leaves the process descriptor at the tail of `runqueue`
      - Ensures a fair assignment of CPU time to all `SCHED_RR` real-time processes with the same priority

    `SCHED_NORMAL` * Conventional time-shared process
  - Scheduling of conventional processes
    - Each conventional process has its own static priority
    - Static priority used by the scheduler to rate the process with respect to other conventional processes in the system
    - Static priority in the range 100 (highest priority) to 139 (lowest priority)
    - A new process inherits the static priority of its parent
      - Static priority can be changed by passing a *nice value* using `nice(2)` or `setpriority(3C)`

- Base time quantum Q_b (in ms)
  - Based on a function of static priority P_s

$$Q_b = \begin{cases} (140 - P_s) \times 20 & \text{if } P_s < 120 \\ (140 - P_s) \times 5 & \text{if } P_s \geq 120 \end{cases}$$

  - Higher the static priority (lower its numerical value), the longer the base quantum
- Dynamic priority P_d and average sleep time
  - Ranges from 100 (highest priority) to 139 (lowest priority)
  - Number used by scheduler when selecting a new process to run

$$P_d = \max(100, \min(P_s - b + 5, 139))$$

    - b is a bonus value ranging from 0 to 10
    - $b < 5$ is a penalty to lower P_d
    - $b > 5$ raises P_d
    - b depends on the past history of the process; related to average sleep time of the process
  - Average sleep time
    - Given by the average number of nanoseconds spent by the process in sleep state
    - Sleeping in `TASK_INTERRUPTIBLE` state contributes to average sleep time in a different way from sleeping in `TASK_UNINTERRUPTIBLE` state
    - Average sleep time decreases while a process is running
    - Can never become larger than 1s

Avg sleep time	Bonus	Granularity
0ms $\leq T_s <$ 100ms	0	5120
100ms $\leq T_s <$ 200ms	1	2560
200ms $\leq T_s <$ 300ms	2	1280
300ms $\leq T_s <$ 400ms	3	640
400ms $\leq T_s <$ 500ms	4	320
500ms $\leq T_s <$ 600ms	5	160
600ms $\leq T_s <$ 700ms	6	80
700ms $\leq T_s <$ 800ms	7	40
800ms $\leq T_s <$ 900ms	8	20
900ms $\leq T_s <$ 1000ms	9	10
1s	10	10

    - Average sleep time also used to determine if a process is considered interactive or batch

      A process is interactive if it satisfies

$$P_d \leq 3 \times P_s/4 + 28$$

      which is equivalent to

$$b - 5 \geq P_s/4 - 28$$

- Interactive delta Δ_i
  - Given by $P_s/4 - 28$
  - Easier for higher priority processes to become interactive
  - A process with highest P_s (100) is considered interactive if its $b > 2$, or its average sleep time > 200ms
  - A process with lowest P_s (139) is never considered interactive because b is always < 11 which is required to reach $\Delta_i = 6$
  - A process with default $P_s = 120$ becomes interactive as soon as its average sleep time exceeds 700ms
- Sleep threshold Θ_s

Priority values for a conventional process					
Description	P_s	Nice val	Q_b	Δ_i	Θ_s
Highest static priority	100	-20	800ms	-3	299ms
High static priority	110	-10	600ms	-1	499ms
Default static priority	120	0	100ms	$+2$	799ms
Low static priority	130	$+10$	50ms	$+4$	999ms
Lowest static priority	139	$+19$	5ms	$+6$	1199ms

  - Active and expired processes
    - Processes with higher P_s get larger slice of CPU time but they should not completely lock out processes with lower P_s
    - Process starvation is avoided by scheduling a lower priority process whose time quantum has not been exhausted when a higher priority process finishes its quantum
    - Active processes: Runnable processes that have not exhausted their time quantum and are allowed to run
    - Expired processes: Runnable processes that have exhausted their time quantum and are forbidden to run until all active processes expire

- Scheduling of real-time processes
  - Real-time priority ranges from 1 (highest) to 99 (lowest)
  - Scheduler always favors a higher priority runnable process over a lower priority one
  - Real-time processes always considered active
  - If there are several real-time runnable processes at the same priority, scheduler chooses the process that occurs first in the corresponding list of local CPU's runqueue

- Data structures used by scheduler
  - `runqueue`
    - Each CPU has its own `runqueue`
    - Every runnable process in the system belongs to exactly one `runqueue`
    - Runnable processes may migrate from one runqueue to another, to move to a different CPU
    - Each structure also contains two `arrays` of 140 doubly linked list heads
      - One list corresponding to each priority (0 to 139)
      - `arrays[0]` contains expired processes
      - `arrays[1]` contains active processes
      - The role of the two `arrays` changes periodically; active processes become expired while expired processes become active

- Allocating time slices to children
  - Both parent and child get half the number of ticks left to the parent
  - Done to prevent children from getting unlimited amount of CPU time
    - Parent creates a child that runs the same code and kills itself
    - If creation rate is adjusted properly, the child can get a full quantum before the parent's quantum expires
    - A continutation of the above allows infinite time to a single process family
  - This also prevents a process to use excessive time by starting several processes in the background