

## 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)
- 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
  - \* Swapping system: Writes entire process to the swap device
  - \* Paging system: Writes pages of memory to the swap device
- Allocates secondary memory for efficient storage and retrieval of user data
- Allows controlled peripheral device access to processes
- 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

### 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 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.
    - I/O status information. Outstanding I/O requests, I/O devices allocated, list of open files, etc.
- 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  $\Rightarrow$  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
  - Includes items such as program counter, registers, and stack pointer

### 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 the above 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
    - \* `r0`. Initial allocation of resources
    - \* `k0`. Initial priority of the process
    - \* Accounting information and limits
    - \* Add the process to the *ready list*
    - \* Initial allocation of memory and resources must be a subset of parent’s and be assigned as shared
    - \* Initial priority of the process can be greater than the parent’s
  - `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  $P_1$

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

- Two processes  $P_0$  and  $P_1$  without multitasking

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

- Processes  $P_0$  and  $P_1$  with multitasking

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

- 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*
  - 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  $\Rightarrow$  Less swapping overhead
  - Smaller memory size  $\Rightarrow$  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  $\Rightarrow$  Higher priority
  - Batch processes  $\Rightarrow$  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 Scheduling
    - 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

– Priority function  $P(r) = r$

• Last-in/First-out 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.

– Priority function  $P(r) = -r$

• Shortest Job Next 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

– Priority function  $P(t) = -t$

– 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

• Longest Job First Scheduling – Homework exercise

• 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
- Priority function  $P(\tau) = a - \tau$ , where  $a$  is the arrival time
- 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
- Preemption based on fixed time slices or time quanta  $q$
- Time quantum between 10 and 100 milliseconds
- All user processes treated to be at the same priority
- Ready queue treated as a circular queue
  - \* 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

- Let there be  $n$  processes in the ready queue, and  $q$  be 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
- 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  $T_{\Pi}$  time units at priority level  $\Pi$
- 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  $T_{\Pi}$ , 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
- *Aging* to prevent starvation

- 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
- BeOS has a scheduling latency of 250 ms, making it suitable for many real-time applications, and improves responsiveness for non-real-time applications
- Unix processes cannot be preempted when they are running in kernel mode; this makes Unix unsuitable for real-time processing