Computer Operating Systems 214

2. PROCESS MANAGEMENT
Copyrights on Figures

• Figures presented in these notes with the following captions are extracted from the textbook and copyrighted by the publisher as indicated

• DISCLAIMER
  – Whilst every attempt is made to maintain the accuracy and correctness of these notes, the author, Dr. Roberto Togneri, makes no warranty or guarantee or promise express or implied concerning the content.
Multi-programming Processes

• Program and Processes
  – *Program*: the executable code
  – *Process*: the program in action (program is “running”)

• N processes, 1 processor
  – *Uni-processor* system
  – Processor switches between process → **pseudo-parallelism**
    • N processes loaded in memory → multi-programming
  – Human analogy
    • write lecture notes, mark lab, prepare tutorials → 3 tasks, but only one of me!
    • phone rings, somebody knocks on door → event/interrupt driven
    • when to switch tasks? “had enough” or “get interrupted”
  – Non-deterministic behaviour
    • process involves a sequence of instruction steps which can be interrupted and resumed at unpredictable times → cannot make assumptions on timing.

• N processes, M processors (M < N)
  – *Multi-processor* system
  – Only M of the N processes run concurrently on M processors
    • processes have to be switched between processors → complex scheduling problem
Process Creation and Termination

• Creating Processes
  – A process (parent) can spawn/create another process (child)
  – OS resources have to be allocated to a new process
    • New entry in the list of processes OS is maintaining (process table)
    • Memory allocated for Process Control Block (PCB) or process descriptor
    • Each process is allocated separate code, stack and data memory segments

• Terminating Processes
  – Process can terminate for a variety of reasons
    • normal completion of instructions
    • execution error or fault (infamous UNIX “segmentation fault”)
    • termination or kill signal
  – OS resources are released when a process is terminated
Process Control Block (*)

• Process Control Block (PCB)
  – OS data structure which contains the state information for each process
    • one PCB per process
  – State information is needed to suspend and correctly resume process execution when another process is scheduled to run
    • Process Identifier (PID), which identifies process
    • User identifier (UID), which identifies the user owning the process
      – User identification is passed from parent to child
    • CPU state
      – Data registers, Program Counter(PC), Stack Pointer(SP), PSW, MAR, MBR, etc.
  • Process Scheduling Control
    – Priority, events pending, process state
  • Process Accounting Information
    – PID, UID, amount of memory used, CPU time elapsed, etc.
    – *Example:* the UNIX `ps` command information
  • Memory Management
    – Location and access state of all user data
  • I/O Management
    – Files and devices currently opened
    – Device buffer status
Process States

- **New**: new process is created
- **Ready**: process is queued waiting to access the CPU
  - OS maintains a *ready queue* of all processes waiting to access the CPU
- **Running**: process is executing
  - Only M processes can be in the Running state for M processors
- **Blocked**: process is waiting for an event and cannot currently execute
  - `read()`, `write()` → disk I/O, mouse, keyboard, display graphics
  - OS maintains a blocked queue for each event or device
- **Exit**: process is terminated
Process Transitions (*)

• **Process Transitions**
  
  – **Dispatch** (Ready → Running): OS has scheduled this process to run
  
  – **Time-out** (Running → Ready): Process has used up its allocated CPU time
    
    • **Pre-emptive scheduling**: timer interrupt is enabled to “time-out” a process
    → mandatory for time-sharing and fair process scheduling
    
    • **Non pre-emptive scheduling**: No timer interrupt .:. process runs until Terminated or Blocked → limited usefulness with modern systems
  
  – **Event Wait / Blocked** (Running → Blocked): Process is waiting for an event
    
    • Allows other processes to access CPU while waiting for an event to happen
    → very efficient!
  
  – **Event Occurs / Wakeup** (Blocked → Ready): Event occurs and process is ready to use CPU again
Interrupt Processing

- Simple Interrupt Processing

![Diagram: Interrupt Processing Flowchart]

Figure 1.10, OS3e
Interrupt Processing

- MINIX Interrupt Processing (for device read)
  - Hardware stacks PC, PSW, etc.
  - Hardware loads the new PC from the interrupt vector associated with the particular interrupt
  - Assembly language procedure saves registers
  - Assembly language procedures sets up new stack
  - C interrupt service runs (typically reads and buffers input)
  - Scheduler( ) marks waiting task as ready
  - Scheduler( ) decides which process is to run next
  - If new process, C procedure saves the current state of the user process
  - C procedure returns to the assembly code
  - Assembly language procedure starts up (new) process
Process Switching

• Process Switching
  – When does a process in the Running state change state?
    • Normal termination or Trap (→ Exit)
      – OS scheduler() (or dispatcher) routine then selects next process to run
    • Interrupt is received (→ Ready)
      – Normal system hardware interrupt processing is invoked to service interrupt. The interrupt handler passes control to the scheduler() which then selects another process or returns control to the current process. The former involves a process switch.
      – Clock interrupt → “time-out” and scheduler() is invoked to select next process
      – I/O interrupt → device interrupt handler is run and then scheduler() selects next process
        • Any processes Blocked on the particular device are moved to the Ready queue
      – Memory fault → process makes an address reference which does not exist, this triggers a special interrupt/trap which invokes the OS memory management routines. This may force the process to terminate.
    • System call (→ Blocked, Ready)
      – Process makes a system call and control is transferred to the OS
      – I/O system call → Usually results in process being moved to the Blocked queue
      – CPU system call → Process moved to Ready queue and OS scheduler( ) selects next process (which could be the same process)
Process Switching

- What happens with a process switch (P1 switched to P2)?
  - P1 state (CPU registers, memory and I/O state, etc.) is saved and PCB1 updated
  - P1 is moved to the appropriate queue (Ready, Blocked)
  - OS scheduler() selects P2 to run next
  - P2 state is loaded from PCB2, and control is passed to P2

- **Context Switch**
  - The overhead involved in switching the current process execution context
    - P1 runs → interrupt handler ; OS scheduler( ) → P1 runs (no process switch)
    - P1 runs → interrupt handler; OS scheduler () ; save PCB1, load PCB2 → P2 runs

- OS scheduler( ) also requires loading/restoring context of scheduler( ) process itself
  - → two context switches per process switch!
- High number of context switches → less CPU time devoted to user

- When does the OS run?
  - After interrupt handler is invoked (including timer interrupt)
  - When user process makes a system call
Thread Concept

• Process Concept
  – Resource ownership
    • data and stack memory, I/O devices, etc.
  – Control execution
    • execution state, sequence of operations

• Thread (Lightweight Process) Concept
  – Same resource ownership, Different control execution
  – Single process with multiple threads of execution

• Figure 2.6, osdi2: (a) 3 processes (b) 1 process with 3 threads
Thread Example and Issues

- Use of threads in a file server
  - Each request is handled by a separate thread
    - Main server process spawns a thread for each request that it receives
  - All threads access the same shared memory area
    - Each thread accesses the same disk cache buffer
      - no need for duplicate buffer management for the same data
  - Threads execute independently of one another
    - If one thread blocks waiting for I/O other threads can continue running

- Issues with Threads
  - User process manages threads
    - OS does need to be thread aware
    - Switching threads is very efficient since this is done within the same user process (no context switching)
    - BUT: thread blocks → OS thinks process blocks → all threads block
  - OS manages threads directly
    - OS can schedule threads directly (if thread blocks, corresponding process does not)
    - BUT: thread switching is more expensive when OS is involved (per thread context switches)

- Example packages
  - POSIX P-threads and MACH C-threads
Case Study: UNIX System V

- Process States

  - **Description**
    - *User running* → Process executing user mode (normal execution)
    - *Kernel running* → Process executing in kernel mode (process makes system call)
    - *Ready to Run, In Memory* → Process is in Ready queue
    - *Preempted* → Special Ready queue when process returns from kernel to user mode, but kernel preempts it and schedules another process
    - *Asleep in Memory* → Process is Blocked waiting for an event
    - *Ready to Run, Swapped* → Process is Ready to run, but is not in main memory
    - *Sleep, Swapped* → Process is Blocked, and is no longer in main memory
Case Study: UNIX System V

• **Process Description**
  • **User-level context**
    – Basic elements which can be generated from program executable/object file
    – *Contents:* Process text, Process data, User stack, Shared memory
  • **Register context**
    – Status information stored about process when it is no longer running
    – *Contents:* Program Counter, Processor Status Register, Stack Pointer, other CPU Registers
  • **System-Level context**
    – Remaining information needed by OS to manage the process
    – Process Table Entry
      • Process state (running, ready, ...), Pointers to process memory area, Size of process
      • User identifiers (User ID, Effective UID)
      • Process identifiers (Process ID, Parent PID)
      • Event descriptor, Priority for scheduling, Signals sent to process
      • Timers (real-time, CPU user time, CPU kernel/system time)
      • P-link pointer to next process in the ready queue
      • Memory status (in memory, swapped to disk, etc.)
    – U (user) Area:
      • Process control information needed when process is running
  • Preprocess Region Table
    • used by memory-management system
  • **Kernel Stack**:
    • special stack used when process is executing in kernel model
Case Study: UNIX System V

• Special Processes
  – *Process 0*
    → created when system boots (PID = 0)
    → *sched* (System V), *swapper* (SunOS), etc.
  – *Process 1*
    → spawned by process 0 as process *init* (PID = 1)
    → all processes have init as their (long-lost?) ancestor
      • init creates initial user process shell when user logs in and system processes (daemons) as system boot scripts are processed
  – *Daemons*
    • Process (usually server) which spends most of its time waiting for a request, or periodically wakes up to perform some function (e.g. lpd, telnetd, inetd, etc.)
  – *Example:*
    • `% ps lax` (PID and PPID fields for init and other processes)

• System Calls
  – *fork()* is used to spawn child processes by cloning the parent
  – *exec(char *file, …)* starts executing the program code in *file*
    • shell process does fork() and child process does exec() to start new process
  – *exit(int status)* terminates process execution and returns *status* to the parent
Case Study: Windows NT

• Characteristics of Windows NT processes
  • NT processes are implemented as objects
  • An executable process may contain one or more threads
  • Both process and thread objects have built-in synchronisation capabilities
  • NT kernel does not maintain relationships (parent-child) among processes it creates

  ![Figure 4.11, OS3e: NT Process and Its Resources]

• Virtual Address Space Description
  – Memory-management system blocks defining memory-allocation of process
• Access Token
  – Process must have a handle to this in order to change its own attributes
• Object Table
  – Contains handles to all other objects known by this process
    • Thread x: one handle for each thread contained by process
    • File y: one handle for each file object
    • Section z: one handle for each section of shared memory
Case Study: Windows NT

- **Windows NT Process Object Attributes**
  - Process ID; Access Token; Base Priority
  - Default processor affinity (default set of processors for threads → multi-processor)
  - Quota Limits on memory and time; Execution time elapsed
  - I/O counters; Virtual Memory (VM) operation counters
  - Exception/Debugging ports (Inter-Process Communication channel when process threads cause an exception)
  - Exit status

- **Windows NT Thread Object Attributes**
  - Thread ID; Thread context (execution state); Base and Dynamic Priority
  - Thread processor affinity; Thread execution time
  - Alert status (should thread execute an asynchronous Remote Procedure Call?)
  - Suspension Count (how many times has thread been suspended?)
  - Impersonation Token (temporary access token assigning alternative access rights)
  - Termination port (Inter-Process Communication channel when thread terminates)
  - Thread Exit Status
Process Scheduling

- 2-level scheduler

- Long-term or High-level Scheduling
  - High-level scheduler is invoked to select which processes:
    - remain in main memory and are subject to short-term scheduling
    - should be released from main memory and **swapped** to disk

- Short-term Scheduling
  - Processes in main memory are placed in Ready queue
  - Short-term scheduler is run to select which process should be moved to the running state
  - Which process should be selected? → **Scheduling Policy**
Short-Term Scheduling

- **Process Execution**
  - Sequence of CPU and I/O Bursts
    - Process execution can be modelled as an alternating series of CPU bursts (normal execution of program instructions) and I/O bursts (process waits for I/O event)
      
      \[
      \text{Process Execution Time} = C_1 + D_1 + C_2 + D_2 + C_3 + D_3 + \ldots
      \]
    
    - **CPU bound** process: \( \sum C_i >> \sum D_i \)
    
    - **I/O bound** process: \( \sum D_i >> \sum C_i \)

- **Arrival Time** (\( T_A \)) and **Service Time** (\( T_S \))
  - **Service Time** = CPU burst time for next scheduled run of process
  - **Arrival Time** = Clock time when process arrives in the ready queue

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>
Short-Term Scheduling Criteria

- **Type of scheduling**
  - **Preemptive:** The OS can force the currently running process to the Ready state
    - Time-out clock interrupt transition
    - Scheduling policy decision after interrupt handler returns
    - Requires clock or time hardware interrupt facility
  - **Nonpreemptive:** Process “runs to completion”
    - Control is returned to OS to schedule next process if current process *terminates* or *blocks* waiting for I/O (completes current CPU burst)

- **Measurable Parameters**
  - **Response Time** ($T_R$)
    - Time elapsed between the initial request and perceived response
    - *Interactive process example:* Time for screen to echo character typed on keyboard
  - **Waiting Time** ($T_w$)
    - Amount of time process spends in the ready queue waiting to access the CPU
  - **Turnaround Time** ($T_Q$) = *Finish Time*(TF) - *Arrival Time*(TA)
    - Time elapsed between submission of process (arrival of process in ready queue) and completion of execution (CPU burst completes)
Short-Term Scheduling Criteria

- Throughput
  - Number of processes completed per unit time
- CPU Utilisation
  - Percentage of time CPU is busy

• Scheduling Policy Decision Criteria
  - Fairness
    - Make sure each process gets its fair share of the CPU
  - Efficiency
    - Maximise the CPU utilisation
  - Response Time
    - Minimise $T_R$
  - Turnaround Time
    - Minimise $T_Q$
  - Throughput
    - Maximise the throughput

• Scheduling Policy Conflicts
  - Minimising $T_R$ for some processes may involve increasing $T_Q$ for others
Short-Term Scheduling Policies

- **First-Come, First-Server (FCFS)**
  - Oldest process in ready queue (in terms of time elapsed since arrival) is selected as the next process to run to completion

<table>
<thead>
<tr>
<th>Process</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival Time ($T_A$)</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Service Time ($T_S$)</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FCFS Finish Time ($T_F$)</td>
<td>3</td>
<td>9</td>
<td>13</td>
<td>18</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Turnaround Time ($T_Q$)</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td>12</td>
<td>8.60</td>
</tr>
<tr>
<td>$T_Q/T_S$</td>
<td>1.00</td>
<td>1.17</td>
<td>2.25</td>
<td>2.40</td>
<td>6.00</td>
<td>2.56</td>
</tr>
</tbody>
</table>

- **Round-Robin (RR)**
  - FCFS scheduling is used to select next process to run
  - Each process is allowed a maximum amount of time or quantum ($q$) and is subject to preemptive timeout when this expires → **time-slicing**
  - If {CPU burst time} < quantum then RR degenerates to FCFS
  - CPU bound processes access CPU more often then I/O bound processes
    - Poor use of I/O devices, increased response time variance
  - How long should quantum interval be?
    - Too small → too many context switch overheads, but better interactive response
    - Too large → less context switch overheads, but poorer interactive response
RR(q=1) Example

- An arriving process will be placed in the ready queue ahead of the current running process

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival Time (T_A)</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Service Time (T_S)</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RR Finish Time (T_F)</td>
<td>4</td>
<td>18</td>
<td>17</td>
<td>20</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Turnaround Time (T_Q)</td>
<td>4</td>
<td>16</td>
<td>13</td>
<td>14</td>
<td>7</td>
<td>11.00</td>
</tr>
<tr>
<td>T_Q/T_S</td>
<td>1.33</td>
<td>2.83</td>
<td>3.25</td>
<td>2.80</td>
<td>3.50</td>
<td>2.74</td>
</tr>
</tbody>
</table>
RR(q=4) Example

- An arriving process will be placed in the ready queue ahead of the current running process.

| Process | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---------|---------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| Arrival | 2 | 3 | 4 | 5 |               |               |               |               |               |               |               |               |               |               |               |               |               |               |               |               |
| Finish  | 1 |   |   |   |               |               |               |               |               |               |               |               |               |               |               |               |               |               |               |               |

<table>
<thead>
<tr>
<th>Process</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival Time ($T_A$)</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Service Time ($T_S$)</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RR Finish Time ($T_F$)</td>
<td>3</td>
<td>19</td>
<td>11</td>
<td>20</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>Turnaround Time ($T_Q$)</td>
<td>3</td>
<td>17</td>
<td>7</td>
<td>14</td>
<td>9</td>
<td>10.00</td>
</tr>
<tr>
<td>$T_Q/T_S$</td>
<td>1.00</td>
<td>2.83</td>
<td>1.75</td>
<td>2.80</td>
<td>4.50</td>
<td>2.58</td>
</tr>
</tbody>
</table>
Short-Term Scheduling Policies

- **Shortest Job First (SJF also SPN, SJN)**
  - The process with the shortest service time is selected as the next process to run to completion → *yields minimum average turnaround time*
    - Favours processes with short service times (i.e. I/O bound processes)
    - Processes with long service times can be starved
  - **Problem** → Need to know the service time for each process
    - The service time for a process cannot be predicted but can be estimated
    - Use *aging* to predict the service time from previous measurements
      - \( S_{\text{next}} = aT_{\text{curr}} + (1-a)S_{\text{curr}} \), where \( a=0.5 \)
      - \( S_{\text{next}} = \) predicted next service time
      - \( S_{\text{curr}} = \) predicted service time and \( T_{\text{curr}} = \) measured service time, for current run
    - Estimation of service time is a scheduling overhead → not efficient

- **Shortest Remaining Time (SRT also “preemptive SJF”)**
  - The process with the shortest remaining time is selected as the next process
    - Remaining time is the service time of new process admitted to the system or the remaining service time time of the current process running
    - If a new process has a shorter service time than the remaining service time of the currently running process then that process is *preempted*
    - Can outperform SJF
  - The service time needs to be predicted as for SJF
  - Processes with long service times can be starved
### SJF Example

<table>
<thead>
<tr>
<th>Process</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>SJF →</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

#### Arrival

<table>
<thead>
<tr>
<th>Arrival</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

#### Finish

<table>
<thead>
<tr>
<th>Finish</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>

#### Process Table

<table>
<thead>
<tr>
<th>Process</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Service</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SJF Finish</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td>20</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Turnaround</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>14</td>
<td>3</td>
<td>7.60</td>
</tr>
<tr>
<td>T_Q/T_S</td>
<td>1.00</td>
<td>2.17</td>
<td>1.00</td>
<td>2.80</td>
<td>1.50</td>
<td>1.69</td>
</tr>
</tbody>
</table>
## SRT Example

| SJF → | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  | 11  | 12  | 13  | 14  | 15  | 16  | 17  | 18  | 19  | 20  |
|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Process | 1   | X   | X   | X   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 2     |     |     |     |     |     |     |     |     |     |     |     |     |     |     | X   | X   | X   | X   | X   |     |
| 3     |     |     | X   | X   | X   |     |     |     |     |     |     |     |     |     |     |     | X   | X   | X   |     |
| 4     |     |     |     |     |     |     |     |     |     | X   | X   | X   | X   | X   |     |     |     |     |     |     |
| 5     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Arrival | 2   | 3   | 4   | 5   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Finish  | 1   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

<table>
<thead>
<tr>
<th>Process</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival Time ($T_A$)</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Service Time ($T_S$)</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SRT Finish Time ($T_F$)</td>
<td>3</td>
<td>15</td>
<td>8</td>
<td>20</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Turnaround Time ($T_Q$)</td>
<td>3</td>
<td>13</td>
<td>4</td>
<td>14</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$T_Q/T_S$</td>
<td>1.00</td>
<td>2.17</td>
<td>1.00</td>
<td>2.80</td>
<td>1.00</td>
<td>1.59</td>
</tr>
</tbody>
</table>
Short-Term Scheduling Policies

- Highest Response Ratio Next (HRRN)
  - The response ratio is defined as:
    - \( \text{ReR} = (\text{waiting time} + \text{service time}) / \text{service time} \)
  - The process with the highest ReR is selected as the next process to run to completion
    - Favours processes with short service times (i.e. I/O bound processes)
    - As ReR increases for processes with long waiting times such processes will not be starved
  - Calculation of ReR
    - \( \text{ReR} = 1 + (\text{Tw}/\text{T}s) \)
    - P1 is scheduled first, then P2.
      After P2 finishes (t=9), all processes are in the ready queue for (t=10)
      - P3: \( (\text{Tw}/\text{T}s) = (6/4) = 1.5 \)
      - P4: \( (\text{Tw}/\text{T}s) = (4/5) = 0.8 \)
      - P5: \( (\text{Tw}/\text{T}s) = (2/2) = 1.0 \)
        Hence P3 is selected next
    - P3 finishes at (t=13), we have for (t=14):
      - P4: \( (\text{Tw}/\text{T}s) = (8/5) = 1.6 \)
      - P5: \( (\text{Tw}/\text{T}s) = (6/2) = 3.0 \)
        Hence P5 is selected next, and then P4

<table>
<thead>
<tr>
<th>Process</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival Time (( T_A ))</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Service Time (( T_S ))</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
### HRRN Example

#### Problem with SJF, SRT and HRRN

- Requires estimation of service time

#### Problem with RR

- Favours CPU bound processes

<table>
<thead>
<tr>
<th>Process</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finish</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Service</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>HRRN</td>
<td>3</td>
<td>9</td>
<td>13</td>
<td>20</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Turnaround</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>14</td>
<td>7</td>
<td>8.00</td>
</tr>
<tr>
<td>T/Ts</td>
<td>1.00</td>
<td>1.17</td>
<td>2.25</td>
<td>1.80</td>
<td>6.00</td>
<td>2.44</td>
</tr>
</tbody>
</table>
Short-Term Scheduling Policies

• Priority Scheduling (Feedback)
  – Multi-level ready queues: RQ0, RQ1, RQ2, …
  – Different priority scheduling for each queue
    • If i < j then RQi is a higher priority to RQj
    • RQ0 is highest priority queue
  – Scheduler selects processes from highest queue which is non-empty
    • Select next process from RQ0,
      If RQ0 is empty then select process from RQ1
      If RQ1 is empty then select process from RQ2, and so on
    • Processes in a higher priority queue will always be scheduled in favour of processes in lower priority queues
  – Process priority aging using priority-based RR scheduling
    • Operation
      – Each new process is admitted to RQ0
        If process runs and times-out it is moved to RQ1
        else process blocks before expiry of quantum and is returned to RQ0 upon wakeup
      – As process keeps timing out it is moved down each priority queue
        If process begins to block before time-out it can be moved up each priority queue
    • CPU bound process will have lower priority to I/O bound processes
Short-Term Scheduling Policies

- CPU bound processes can experience long turnaround times due to lower priority
  - Compensate by allowing lower priority processes to run for longer:
    e.g. RQi has a quantum of length $2^i$

Figure 9.4, OS3e: *Priority Scheduling*
Real-time Scheduling

• Real-time concept
  – *Plant process control*: React to real-time sensor or actuation events
  – *Multimedia*: Acquire or playback digital data at a constant rate
  – **hard real time**: absolute deadline that must be met
    • Car must turn right or else it will go off the road
  – **soft real time**: missing occasional deadline is acceptable
    • Multimedia playback can safely skip a frame or audio interval

• Problem with current scheduling policies
  – No guarantee of when the process will be scheduled next

• Real-time solutions
  – Specially designed real-time operating system
    • Rapid context switching, fast interrupt handling, etc. → reduce context overhead
  – Real-time scheduling strategies
    • Static table-driven approaches
      – Assumes knowledge of task execution time, arrival time and deadlines
      – Develops a static schedule of task execution in order to meet all deadlines
      – Non-preemptive scheduling if starting deadline is specified (task must start by the specified time and run to completion)
      – Preemptive scheduling if completion deadline is specified (task must complete its execution by the specified time)
Real-time Scheduling

- **Static Priority-Driven Preemptive Scheduling**
  - Makes use of standard preemptive priority scheduling of non real-time systems
  - Priorities are assigned in relation to the deadlines that have to be met rather than whether a task is CPU or I/O bound.

- **Dynamic Planning-Based Scheduling**
  - Task execution schedule adapts to new arrivals (dynamic)

- **Dynamic Best Effort Scheduling**
  - When task arrives its priority is assigned based on the characteristics of the task
  - Used by most commercial systems because it is easy to implement
  - Does not provide any guarantees on performance

  - Real-time scheduling policies
    - **Definition:** A periodic event executes for $C$ seconds every $T$ [frequency $(1/T)$]
    - **Rate Monotonic:**
      - For periodic events assign higher priority in direct proportion to frequency
      - For $n$ tasks this will work if: $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$
      - Used in priority-driven strategies
    - **Earliest Deadline:** Select process with the earliest completion deadline to meet
      - Used in table-driven or planning-based scheduling strategies
    - **Least Laxity:** Select process with the smallest amount of time left
      - If process requires $C$ msec and its deadline is in $D$ msec, the laxity is $(D-C)$ msec
Case Study: UNIX Process Scheduling

- UNIX System V Fair-Share Scheduling Mechanism
  - Multi-level feedback priority queues (up to 32) with RR scheduling
  - Process priorities are recomputed in 1 second intervals
    - Parameters:
      - $P(j,i) =$ Priority number of process $j$ at the beginning of interval $i$ (calculated)
        lower $P(j,i) \rightarrow$ higher priority
      - $Base(j) =$ Base priority of process $j$ (parameter)
        → Process assigned base priority from fixed bands of priority levels:
          - Swapper (highest priority)
          - Block I/O device control
          - File manipulation
          - Character I/O device control
          - User processes (lowest priority)
      - $U(j,i) =$ CPU use of process $j$ in interval $i$ (measured)
      - $CPU(j,i) =$ exponentially weighted average of $U(j,i)$ (calculated)
      - $nice(j) =$ User-controllable adjustment factor (parameter)
        → nice( ) or renice ( ) system calls
    - Formulas:
      - $P(j,i) = Base(j) + 0.5 \times CPU(j,i-1) + nice(j)$
      - $CPU(j,i) = 0.5 \times U(j,i) + 0.5 \times CPU(j,i-1)$
  - CPU bound processes are subject to priority degradation as $P(j,i)$ increases
  - Example: Do ps lax and check PRI field
Case Study: Windows NT Process Scheduling

- **Windows NT Priority Classes**
  - Priority-driven Pre-emptive scheduling
    - When a thread with a higher priority number (than the currently running thread) becomes ready, the currently running thread is pre-empted in favour of the higher priority thread
    - Threads with equal priority are RR scheduled
  - (16-31): Fixed-Priority Real-Time class
    - Thread priorities remain fixed and do not change
  - (0-15): Variable-Priority class
    - Thread priorities are dynamic within the range of this class (a priority 15 thread can never be promoted to a real-time priority of 16)
    - Process Base Priority → initial priority assigned to process
    - Thread Base Priority → initial thread priority relative to Process Base Priority
    - Thread Dynamic Priority → variable priority of running thread
  - Changing variable-priorities by NT executive
    - If thread uses up its current time quantum → priority is lowered
    - If thread blocks for I/O → priority is raised
      - Priority is raised higher for interactive I/O waits (keyboard, mouse, display) than non-interactive I/O waits (disk)
Client-Server Systems

• Definitions
  – Split application across different computing platforms dedicated to the particular functions of server and client
    • Server: high-end computer (large disk capacity, fast I/O, large amount of memory, possible multi-processor) which performs the disk/CPU intensive functions
    • Client: low-end computer which performs the data input formatting and data output presentation and display
  – A client sends a request to a server, and server returns a response

• Applications
  – Database Applications
    • Server: maintains the (huge) database file and backup systems and processes the queries
    • Client: handles the human-computer interface to formulate, present and display database queries to the human user
  – File Server
    • Server: houses the (usually) large, fast disk I/O subsystem, processes all read/write requests and maintains a reliable backup system
    • Client: user workstation which usually presents user with the local filesystem semantics (drive, directory) to access data remotely stored on the file server.
Client-Server Systems

- Many clients, One Server
  - Single-Threaded Server
    - Single server process listens for a client request
    - When client makes a request the single server process responds, any other client requests are placed on hold (queued) until server process completes current request
    - When busy, client requests may time-out waiting for server to respond
  - Multiple-Threaded Server
    - Single server process listens for a client request
    - When client makes a request the server spawns another process (or thread) to handle that request and immediately listens for the next client request
    - When busy, server can create many processes/threads and exploit multi-programming for improved CPU utilisation
Distributed Message Passing

- Inter-Process Communication (IPC)
  - Send Process (Id=222) → Send (ProcessId=111, Message)
  - Receiving Process (Id=111) → Receive (ProcessId=222, Message)

- Issues
  - Reliable or Unreliable?
    - Reliable → guarantees delivery over an unreliable communication network, but this may incur overheads (keeping track of lost data, resending lost data, etc.)
    - Unreliable → no guarantee of delivery but minimal overhead
  - Blocking(Synchronous) or Non-Blocking(Asynchronous)?
    - Blocking → process suspended/blockaded while waiting for send/receive to complete
      - Why Synchronous? Processes can “wait for one another” (i.e. synchronise)
    - Non-Blocking → process does not block
      - send/receive process has to poll OR is signalled when send/receive complete

Figure 13.16, OS3e: Basic message-passing primitives
Remote Procedure Call (RPC)

- Client-server interaction by message passing → **RPC abstraction**
  - Like a normal procedure call, except that procedure is on a remote system!
  - **CALL Remote_Procedure** (Passed Arguments, Returned Values)
    - Arguments are sent from Calling Process to Remote Called Process
    - Values are returned to Calling Process from Remote Called Process

- Figure 13.17, OS3e: *Remote Procedure Call Mechanism*
Remote Procedure Call (RPC)

- Standardised interface for heterogeneous systems and networks
  - Parameters are **packed** to avoid data presentation effects
    - Endian byte ordering of CPU
      - Little Endian → Intel, DEC Alpha
      - Big Endian → SPARC, MIPS
    - Network byte order and Host byte order
      - Network byte order is BigEndian

- Figure 12.25, OS2e: *Remote Procedure Call logic*
Case Study: UNIX IPC Mechanisms

- Processes need to communicate with another
  - Network communication (e.g. telnet, ftp, WWW)
  - Synchronisation of activity (for distributed / parallel processing)

- UNIX pipe( ) system call
  - Single process establishes communication pipe for communication between spawned child processes → limited form of inter-process communication

- UNIX socket( ) abstraction
  - Connection establishment by client-server
    - Server process code
      - preamble code to setup communication parameters
      - listen( ); s = accept( )
    - Client process code
      - preamble code to setup communication parameters
      - s = connect( )
  - Data communication using file I/O semantics
    - write(s, …) to send message to other process
    - read(s, …) to receive a message from other process
Case Study: UNIX IPC Mechanisms

- UNIX domain sockets (local inter-process communication)
  - Server and client processes use an identical *special socket file* to establish channel for communication

- UNIX INET sockets (remote inter-process communication)
  - Server processes establishes a communication PORT on which it listens
  - Client process connects to server on host with Internet (INET) address A.B.C.D listening on PORT
    - s = connect(A.B.C.D, PORT)
  - Used by Internet applications (reserved PORT)
    - telnet server process listens on PORT 23
      (e.g. telnet us1 \(\rightarrow\) connect to server listening on PORT=23, HOST=130.95.200.20)

- UNIX signal( ) mechanism
  - Software version of hardware interrupt mechanism
  - Allows Non-blocking (Asynchronous) communication
  - Process uses *signal*(SIGNUM, sighandler( )) to associate sighandler( ) routine with signal SIGNUM \(\rightarrow\) when process receives SIGNUM it immediately calls sighandler( ) and then returns
    - *signal*(SIGIO, read_data( )) for non-blocking read(s, … )