### UNIT -2

##### PART-1: PROCESS CONCEPT

**Introduction**

An operating system executes a variety of programs: Batch system – jobs

Time-shared systems – user programs or tasks

Textbook uses the terms *job* and *process* almost interchangeably

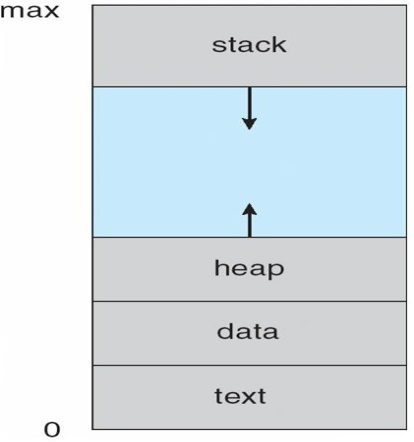
**Process**

* It is a program in execution; process execution must progress in sequential fashion
* A process is more than the program code, which is sometimes known as the **text section**

A process includes:

* **program counter** it consists of the current activity and the contents of the processor's registers
* **stack**, which contains temporary data (such as function parameters, return addresses, and local variables)
* **data section**, which contains global variables
* **heap**, which is memory that is dynamically allocated during process run time.

##### Figure: Process in Memory

****

Is process is equal to program?

* program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file)
* process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources
* A program becomes a process when an executable file is loaded into memory.
  + Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a. out.)

**Process State:**

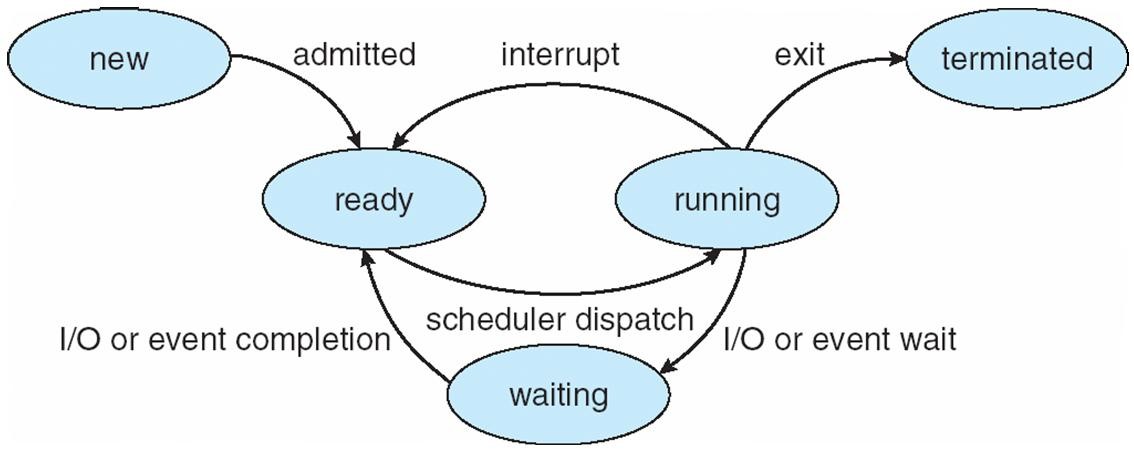
As a process executes, it changes *stat*e

**new**: The process is being created

**running**: Instructions are being executed

**waiting**: The process is waiting for some event to occur **ready**: The process is waiting to be assigned to a processor **terminated**: The process has finished execution

##### Diagram of Process State



**Process Control Block (PCB)**

Each process is represented in the operating system by a process control block (PCB)-also called a task control block

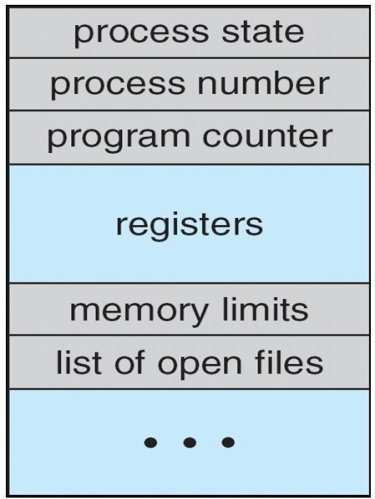
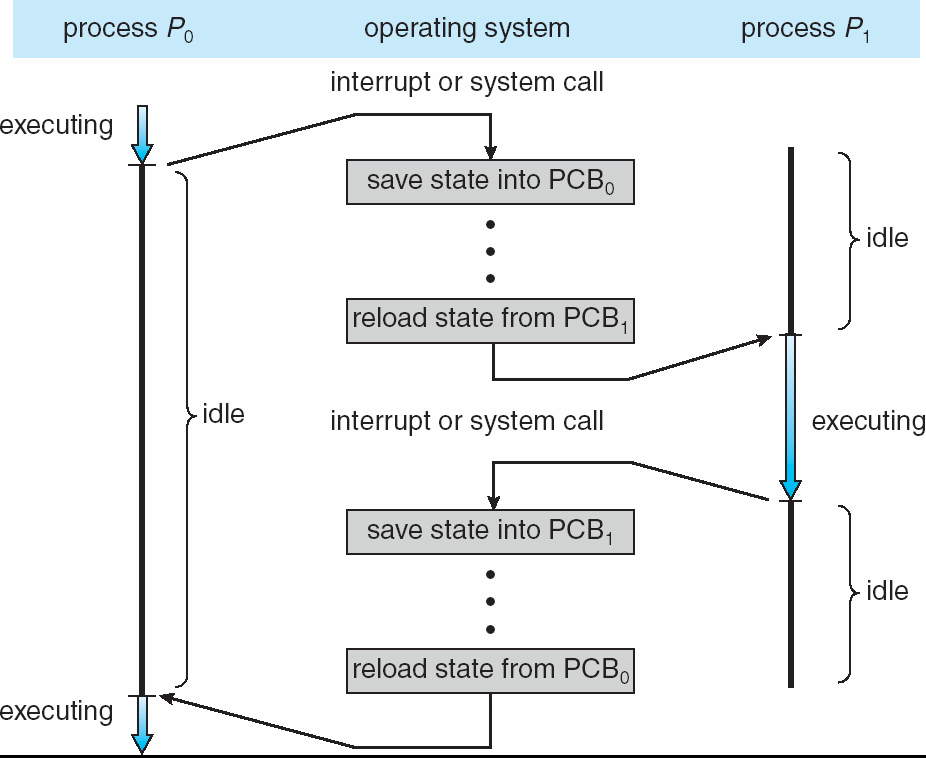


Figure: PCB

Information associated with each process

* **Process state**: The state may be new, ready running, waiting, halted, and so on
* **Program counter**: The counter indicates the address of the next instruction to be executed for this process.
* **CPU registers**: They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Fig a)
* **CPU scheduling information**: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
* **Memory-management information**: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system
* **Accounting information**: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on
* **I/O status information**: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

##### Figure a: CPU Switch From Process to Process



**Operations on Processes**

##### Process Creation:

* **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
* Generally, process identified and managed via **a process identifier** (**pid**) Resource sharing

##### Figure: A tree of processes on a typical Solaris

User login screen

Memory and file systems

Networking services

Above figure illustrates a typical process tree for the Solaris operating system, showing the name of each process and its pid.

* Process needs certain **resources** (CPU time, memory, files, I/O device)
* When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or from the parent process.
* Parent and children share all resources
* Parent and child share no resources because any of a process from overloading the system by creating too many subprocesses.
* When a process creates a new process, two possibilities exist in terms of execution:
* Parent and children execute concurrently
* Parent waits until children terminate
* There are also two possibilities in terms of the address space of the new process
* Child process is a duplicate of parent process
* Child process has a new program loaded into it
* UNIX examples

**fork()** system call creates new process

**exec()** system call used after a **fork()** by one of 2 process to replace the process memory space with a new program

**C Program Forking Separate Process**

int main()

{

pid\_t pid;

/\* fork another process \*/ pid = fork();

if (pid < 0) { /\* error occurred \*/ fprintf(stderr, "Fork Failed"); exit(-1);

}

else if (pid == 0) { /\* child process \*/ execlp("/bin/ls", "ls", NULL); //execlp: execute system call,

//The child process then overlays its address space with the UNIX command /bin/ls

}

else { /\* parent process \*/

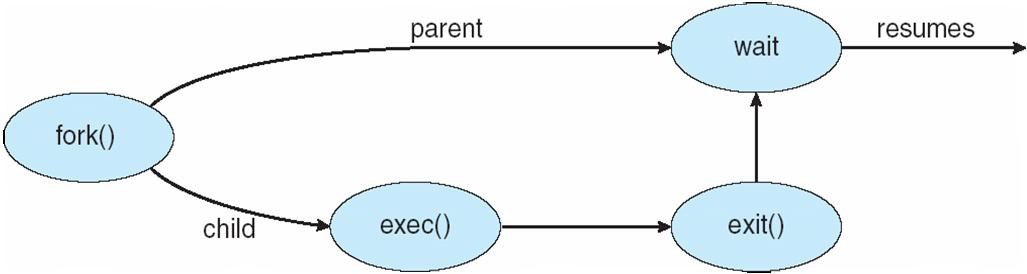
/\* parent will wait for the child to complete \*/ wait (NULL);

printf ("Child Complete"); exit(0);

}

}

##### Process Creation



**Process Termination:**

* Process terminates when executes last statement and asks the operating system to delete it (exit())
* Output data from child to parent (via wait())
* Process resources (physical and virtual memory, open files, and I/0 buffers) are deallocated by operating system
* Parent may terminate execution of children processes (abort()), if child has exceeded allocated resources
* A parent may terminate the execution of one of its children for a variety of reasons:
* The child has exceeded its usage of some of the resources that it has been allocated.
  + Task assigned to child is no longer required
  + If parent is exiting, some operating system do not allow child to continue if its parent terminates
  + If process terminates, then all children must be terminated - cascading termination

##### Interprocess Communication

* Processes executing concurrently in the operating system may be independent or cooperating
* Independent process cannot affect or be affected by the other processes executing in the system.
* Cooperating process can affect or be affected by other processes, including sharing data
* Reasons for cooperating processes:
* **Information sharing**: Since several users may be interested in the same piece of information (shared file), we must provide an environment to allow concurrent access to such information.
* **Computation speedup**: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Speedup can be achieved with multiple processing elements (CPUs or I/O channels)
* **Modularity**: System can be constructed in a modular fashion, dividing the system functions into separate processes or threads
* **Convenience**: Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.
  + Cooperating processes need **interprocess communication (IPC)** mechanism that will allow them to exchange data and information.
* Two models of IPC are Shared memory and Message passing

##### Figure: Communications Models

##### (a) Message passing (b) Shared memory

##### Shared-Memory Systems:

##### Producer-Consumer Problem

* Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
* Example: Producer(web server) produces resources(like HTML files, images, etc) and Consumer(client) consumes by reading it
* Two types of buffers are:
* unbounded-buffer places no practical limit on the size of the buffer
* bounded-buffer assumes that there is a fixed buffer size

##### Bounded-Buffer – Shared-Memory Solution

#define BUFFER\_SIZE 10 typedef struct {

. . .

} item;

item buffer[BUFFER\_SIZE]; int in = 0;

int out = 0;

Here, we can use BUFFER\_SIZE-1 items(eg: 10-1=9)

The shared buffer is implemented as a circular array with two logical pointers: in and out.

The variable **in** points to the next free position in the buffer and **out** points to the first full position in the buffer

##### Bounded-Buffer – Producer

item nextProduced;

while (true) {

/\* Produce an item \*/

while (((in = (in + 1) % BUFFER SIZE) == out) // buffer is full

; /\* do nothing -- no free buffers \*/ buffer[in] = nextProduced;

in = (in + 1) % BUFFER SIZE;

}

##### Bounded Buffer – Consumer

item nextConsumed;

while (true) {

while (in == out)// buffer is empty

; // do nothing -- nothing to consume

// remove an item from the buffer nextConsumed = buffer[out];

out = (out + 1) % BUFFER SIZE;

//return item;

}

##### Message Passing Systems:

* Mechanism for processes to communicate and to synchronize their actions
* Message system – processes communicate with each other without resorting to shared variables
* a **chat** program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages
* Message Passing facility provides two operations:
* **send**(*message*) – message size fixed or variable
* **receive**(*message*)
* If processes P and Q wish to communicate, they need to: establish a communication link between them exchange messages via send/receive
* Implementation of communication link physical (e.g., shared memory, hardware bus) logical (e.g., several methods for logically implementing a link) and the send () I receive() operations:
* Direct or indirect communication
* Synchronous or asynchronous communication
* Automatic or explicit buffering

**Naming-**

##### Direct Communication

* Processes must name each other explicitly:
* **send** (*P, message*) – send a message to process P
* **receive**(*Q, message*) – receive a message from process Q
* Properties of communication link:
* Links are established automatically
* A link is associated with exactly two processes
* Between each pair there exists exactly one link
* This scheme exhibits *symmetry* in addressing; that is, both the sender process and the receiver process must name the other to communicate.
* A variant of this scheme employs *asymmetry* in addressing
* Here, only the sender names the recipient; the recipient is not required to name the sender.
  + send(P, message) -Send a message to process P.
  + receive (id, message) -Receive a message from any process; the variable
* *id* is set to the name of the process with which communication has taken place.
* The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions.

##### Indirect Communication

* Messages are directed and received from mailboxes (also referred to as ports)
* Each mailbox has a unique id
* Processes can communicate only if they share a mailbox
* **send**(*A, message*) – send a message to mailbox A
* **receive**(*A, message*) – receive a message from mailbox A
* Properties of communication link:
* Link established only if processes share a common mailbox
* A link may be associated with many processes
* Each pair of processes may share several communication links
* Link may be unidirectional or bi-directional
* Primitives are defined as:
* Mailbox sharing :P1, P2, and P3 share mailbox A
* *P1*, sends a message to A; while *P2* and *P3* receive
* Who gets the message first?

Solutions:

* Allow a link to be associated with at most two processes
* Allow only one process at a time to execute a receive operation
* Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.
  + - Operations:
* create a new mailbox
* send and receive messages through mailbox
* destroy a mailbox

##### Synchronization-

* + - Communication between processes takes place through calls to send() and receive () primitives
    - Message passing may be either blocking or non-blocking
    - Blocking is considered synchronous
      * Blocking send: The sending process is blocked until the message is received by the receiving process or by the mailbox
      * Blocking receive: The receiver block until a message is available
    - Non-blocking is considered asynchronous
      * Non-blocking send: The sending process sends the message and resumes operation
      * Non-blocking receive: The receiver retrieves either a valid message or null

##### Buffering-

Queue of messages attached to the link; implemented in one of three ways

1. Zero capacity – 0 messages, Sender must wait for receiver
2. Bounded capacity – finite length of n messages, Sender must wait, if the link is full
3. Unbounded capacity – infinite length, Sender never waits

**Scheduling Criteria**

* + - **CPU utilization** – keep the CPU as busy as possible
    - **Throughput** – number of processes that complete their execution per time unit
    - **Turnaround time** – amount of time to execute a particular process/ Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/0
    - **Waiting time** – amount of time a process has been waiting in the ready queue/ Waiting time is the sum of the periods spent waiting in the ready queue
    - **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
    - It is desirable to Max CPU utilization, Max throughput, Min turnaround time, Min waiting time, Min response time
    - An accurate illustration should involve many processes, each a sequence of several hundred CPU bursts and I/O bursts. For simplicity, though, we consider only one CPU burst (in milliseconds) per process in our examples.

**Scheduling algorithms**

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

##### First-Come, First-Served (FCFS) Scheduling:

* The process that requests the CPU first is allocated the CPU first.

Process Burst Time(the duration for which a process gets control of the CPU)

*P1* 24

*P2* 3

*P3* 3

Suppose that the processes arrive in the order: **P1, P2, P3**

**Throughput** for P1= 24, P2= 3, P3= 3

**Average Throughput** = (24+3+3)/3= 10 milliseconds

*[Gantt Chart: which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes]*

The Gantt Chart for the schedule is:



**Waiting time** for P1 = 0; P2 = 24; P3 = 27

**Average waiting time**: (0 + 24 + 27)/3 = 17 milliseconds

**Turnaround time** for P1= 24, P2=27, P3= 30

**Average Turnaround time**: (24+27+30)/3= 27 milliseconds

**Response time:** It is the time difference between arrival time and the first execution time (i.e) in interactive systems where you issue a command and wait the beginning of the output

Suppose that the processes arrive in the order: **P2, P3, P1**

**Throughput** for P1= 24, P2=3, P3=3

**Average Throughput** = (24+3+3)/3 = 10 milliseconds

The Gantt chart for the schedule is:



**Waiting time** for P1 = 6, P2 = 0, P3 = 3

**Average waiting time**: (6 + 0 + 3)/3 = 3 milliseconds

**Turnaround time** for P1= 3, P2=6, P3= 30

**Average Turnaround time**: (3+6+30)/3 = 13 milliseconds

Much better than previous case

* **Convoy effect** as short process waits for the long process to get off the CPU
* The FCFS scheduling algorithm is nonpreemptive (i.e) no interrupt occurs in the scheduling process

Process Arrival time Burst Time

P1 0 9

P2 1 6

P3 312

P4 5 5

**Throughput** for P1= 9, P2=6, P3=12, P4=5

**Average Throughput** = (9+6+12+5)/4= 8 milliseconds

##### Gantt chart

##### 

**Waiting time** for P1 = 0, P2 = 9-1 = 8, P3 = 15-3 = 12, P4 = 27-5 = 22

**Average waiting time**: (0 + 8 + 12+22)/4 = 10.5 milliseconds

**Turnaround time** for P1= 9-0 = 9, P2=15-1 = 14, P3= 27-3 = 24, P4=32-5 = 27

**Average Turnaround time**: (9+14+24+27)/4 = 18.5 milliseconds

##### Shortest-Job-First (SJF) Scheduling:

* Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
* SJF is optimal – gives minimum average waiting time for a given set of processes
* The SJF algorithm can be either preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing.
* The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
* A preemptive SJF algorithm will preempt the currently executing process where as a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.

##### Shortest-Job-First (SJF) Scheduling (Non-preemptive):

|  |  |  |
| --- | --- | --- |
| Process |  | Burst Time |
| *P1* |  | 6 |
| *P2* |  | 8 |
| *P3* |  | 7 |
| *P4* |  | 3 |

**Average Throughput** = (6+8+7+3)/4= 6 milliseconds

|  |  |  |
| --- | --- | --- |
| Process |  | Burst Time |
| *P4* |  | 3 |
| *P1* |  | 6 |
| *P3* |  | 7 |
| *P2* |  | 8 |

****

**Average waiting time** = (3 + 16 + 9 + 0) / 4 = 7 milliseconds

**Average Turnaround time**: (9+24+16+3) / 3 = 13 milliseconds

Now, we need to know the length of the next CPU request, so that we can arrange the processes in increasing order based on length of a process.

##### Determining Length of Next CPU Burst

1. tn=actual length of nth CPU burst
2. Ʈn+1= predicted value for the next CPU burst
3. α,0<=α<=1
4. Define:



tn contains our most recent information; Ʈn stores the past history

The parameter α controls the relative weight of recent and past history in our prediction

α =0, then Ʈn+1 = Ʈn

Recent history does not count

α =1, then Ʈn+1 = *t*n

Only the most recent CPU burst counts

* Can only estimate the length

 Can be done by using the length of previous CPU bursts, using exponential average

##### Prediction of the Length of the Next CPU Burst

##### Exponential average with α = 1/2 and Ʈo= αt-1= 10.

If we expand the formula, we get: 

Ʈ0+1= αt0+(1-α) αt0-1

Ʈ1+1= αt1+(1-α) αt1-1+(1-α)2 αt1-2

........

ti = 6,4,6,4,13,13,13,…..

Ʈn+1= αtn+(1-α) Ʈn

Ʈ0 = 10

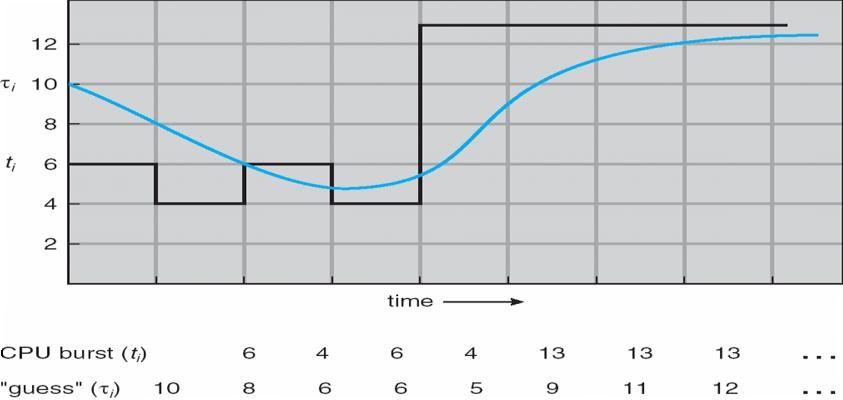
Ʈ0+1= αt0+(1-α) Ʈ0  => Ʈ1 = (½)\*6+(1-1/2)\*10 = 8

Ʈ1+1= αt1+(1-α) Ʈ1 => Ʈ2 = (½)\*4+(1-1/2)\*8 = 6

……..

Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor

##### Figure: Prediction of the length of the next CPU burst

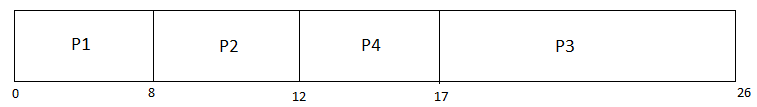


##### Shortest-Job-First (SJF) Scheduling (Non-preemptive):

|  |  |  |
| --- | --- | --- |
| Process | Arrival Time | Burst Time |
| *P1* | 0 | 8 |
| *P2* | 1 | 4 |
| *P3* | 2 | 9 |
| *P4* | 3 | 5 |

**Average Throughput** = (8+4+9+5)/4= 6.5 milliseconds

**Gantt chart**

****

**Waiting time** for P1 = 0, P2 = 8-1 = 7, P3 = 17-2 = 15, P4 = 12-3 = 9

**Average waiting time**: (0+7+15+9)/4 = 7.75 milliseconds

**Turnaround time** for P1= 8-0 = 8, P2=12-1 = 13, P3= 26-2 = 24, P4=17-3 = 14

**Average Turnaround time**: (8+11+24+14)/4 = 14.25 milliseconds

**Preemptive SJF scheduling:** is sometimes called shortest-remaining-time-first scheduling



**Average Throughput** = (8+4+9+5)/4= 6.5 milliseconds

**Gantt chart**

****

**Waiting time** for P1 = 0+(10-1)-0= 9, P2 = 1-1 = 0, P3 = 17-2 = 15, P4 = 5-3 = 2

**Average waiting time**: (9+0+15+2)/4 = 6.5 milliseconds

**Turnaround time** for P1= 17-0 = 17, P2=5-1 = 4, P3= 26-2 = 24, P4=10-3 = 7

**Average Turnaround time**: (17+4+24+7)/4 = 13 milliseconds

* Preemptive SJF scheduling chart average waiting time is 6.5 milliseconds.
* Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

##### Priority Scheduling:

* A priority number (integer) is associated with each process
* The CPU is allocated to the process with the highest priority (smallest integer->highest priority)
* Problem : Starvation – low priority processes may never execute
* Solution : Aging – It is a technique of gradually increasing the priority of the process that wait in a system for long time



**Average Throughput** = (10+1+2+1+5)/4= 6.5 milliseconds

**Gantt chart:**



**Waiting time** for P1 = 6, P2 = 0, P3 = 16, P4 = 18, P5 = 1

**Average waiting time**: (6+0+16+18+1)/5 = 8.2 milliseconds

**Turnaround time** for P1= 16, P2=1, P3= 18, P4=19, P5= 6

**Average Turnaround time**: (16+1+18+19+6)/5 = 12 milliseconds

##### Round Robin (RR) Scheduling:

* Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
* If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
* Performance
* If Time Quantum is large then it acts like FCFS
* If Time Quantum is small then it consists of large with respect to context switch, otherwise overhead is too high

##### Example of RR with Time Quantum = 4

|  |  |
| --- | --- |
| Process | Burst Time |
| *P1* | 24 |
| *P2* | 3 |
| *P3*  The Gantt chart is: | 3 |

****

Typically, hig

her avera

ge turnar

ound than

SJF, but

better *re*

*sponse*

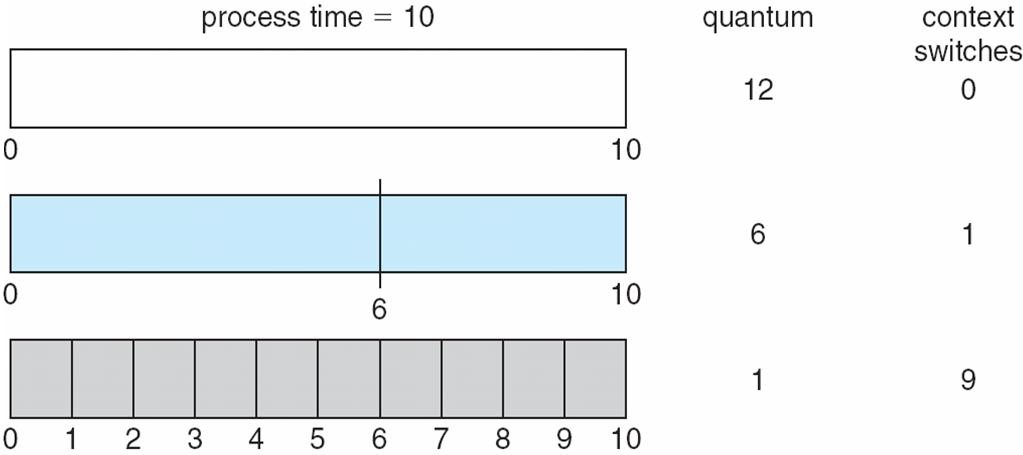
**Waiting time** for P1 =0+10-4=6, P2 = 4, P3 = 7

**Average waiting time**: (6+4+7)/3 = 5.66 milliseconds

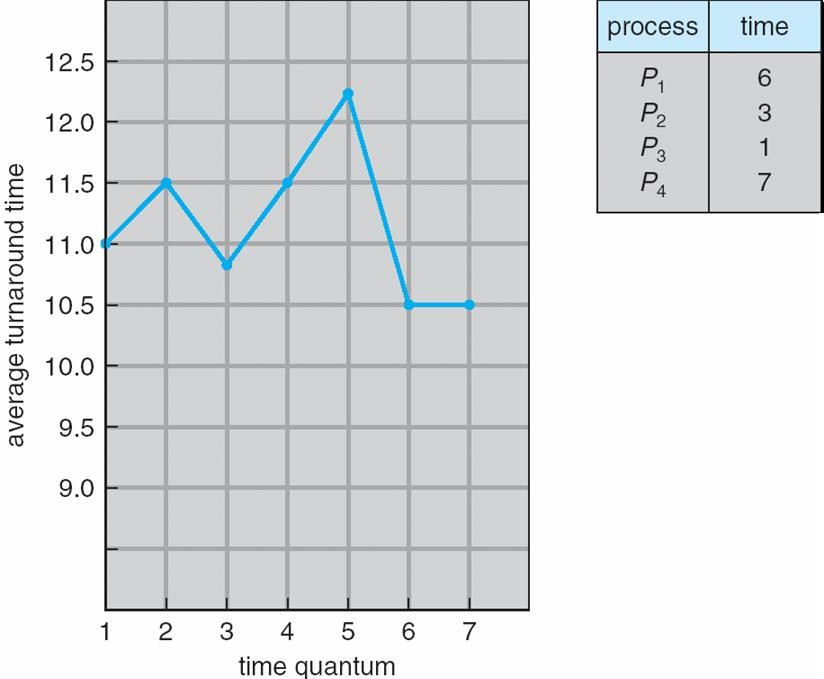
**Turnaround time** for P1= 30, P2=7, P3= 10

**Average Turnaround time**: (16+1+18+19+6)/5 = 15.66 milliseconds

##### Time Quantum and Context Switch Time



**Turnaround Time Varies With The Time Quantum**



* Although the time quantum should be large compared with the context switch time, it should not be too large. If the time quantum is too large, RR scheduling degenerates to an FCFS policy.

##### Multiple-Processor Scheduling

* CPU scheduling more complex when multiple CPUs are available
* Homogeneous processors within a multiprocessor

**Approaches to Multiple-Processor Scheduling:**

**Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing

**Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes.

(*P1-1st processor, P2-2nd processor, problem occurs to retrieve a recent information(P2) from cache memory, cache memory migrate from P1 to P2 to solve this* ***Processor affinity***)

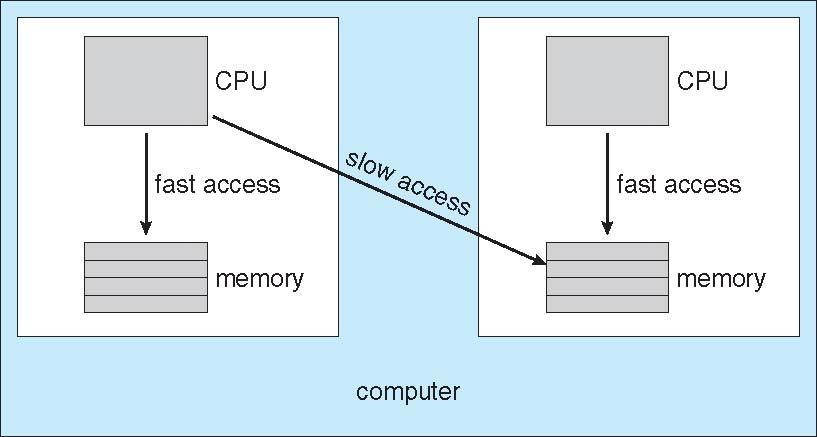
**Processor affinity:**

Process has affinity (which a substance tends to combine with another) for processor on which it is currently running.

**soft affinity** -  is the tendency of a scheduler to try to keep processes on the same CPU as long as possible

**hard affinity –** Linux provides the system call that supports hard affinity, so that no need to migrate

**Figure:** **NUMA(Non Uniform Memory Access) and CPU Scheduling**



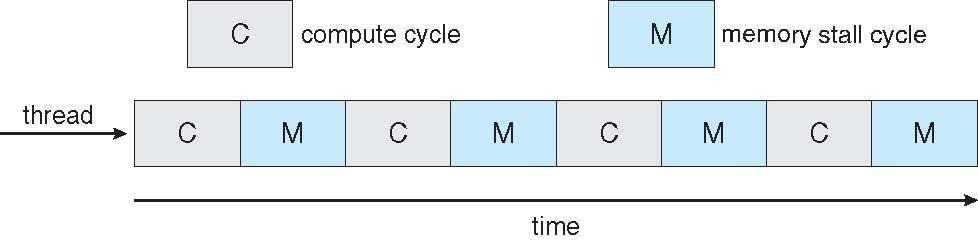
**Load Balancing:**

* Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.
* Load balancing is necessary on systems where each processor has its own private queue of eligible processes to execute.
* On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.
* There are two general approaches to load balancing:
* push migration - a specific task periodically checks the load on each processor and -if it finds an imbalance-evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors
* pull migration - occurs when an idle processor pulls a waiting task from a busy processor.
* Load balancing often counteracts the benefits of processor affinity. That is, the benefit of keeping a process running on the same processor is that the process can take advantage of its data being in that processor's cache memory

**Multicore Processors:**

* Recent trend to place multiple processor cores on same physical chip
* Faster and consume less power
* Multiple threads per core also growing
* When a processor accesses memory, it spends a significant amount of time waiting for the data to become available is called memory stall
* Takes advantage of memory stall to make progress on another thread while memory retrieve happens

##### Figure: Memory stall



In this scenario, the processor can spend up to 50 percent of its time waiting for data to become available from memory.

To remedy this situation, many recent hardware designs have implemented multithreaded processor cores in which two (or more) hardware threads are assigned to each core.

##### Figure: Multithreaded Multicore System

****

From an operating-system perspective, each hardware thread appears as a logical processor that is available to run a software thread. On a dual-threaded, dual-core system, four logical processors are presented to the operating system.

* In general, there are two ways to multithread a processor:
* Coarse-grained multithreading, a thread executes on a processor until a long-latency event such as a memory stall occurs. Because of the delay caused by the long-latency event, the processor must switch to another thread to begin execution. However, the cost of switching between threads is high, as the instruction pipeline must be flushed before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions.
* Fine-grained (or interleaved) multithreading switches between threads at a much finer level of granularity-typically at the boundary of an instruction cycle.

**Virtualization and Scheduling:**

* + The virtualization software presents one or more virtual CPUs to each of the virtual machines running on the system and then schedules the use of the physical CPUs among the virtual machines.
  + Most virtualized environments have one host operating system and many guest operating systems. The host operating system creates and manages the virtual machines, and each virtual machine has a guest operating system installed and applications running within that guest.
  + Each guest operating system may be fine-tuned for specific use cases, applications, and users, including time sharing or even real-time operation

**PART-2: SYNCHRONIZATION**

**Process Synchronization**

* Concurrent( two tasks occurring asynchronously) access to shared data may result in data inconsistency
* Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
* To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
* To present both software and hardware solutions of the critical-section problem
* To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity
* Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer

##### Producer

while (true) {

/\* produce an item and put in nextProduced \*/ while (counter == BUFFER\_SIZE)

; // do nothing

buffer [in] = nextProduced;

in = (in + 1) % BUFFER\_SIZE;

counter++;

}

##### Consumer

while (true) {

while (counter == 0)

; // do nothing

nextConsumed = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

counter--;

/\* consume the item in nextConsumed

}

counter++ could be implemented as

register1 = counter register1 = register1 + 1 counter = register1

counter-- could be implemented as

register2 = counter register2 = register2 - 1 counter = register2

Consider this execution interleaving with “counter = 5” initially:

T0: producer execute register1 = counter {register1 = 5}

T1: producer execute register1 = register1 + 1 {register1 = 6}

T2: consumer execute register2 = counter {register2 = 5}

T3: consumer execute register2 = register2 - 1 {register2 = 4}

T4: producer execute count = register1 {counter = 6 }

T5: consumer execute count = register2 {counter = 4}

* "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full.
* This is an incorrect state because we allowed both processes to manipulate the variable counter concurrently.
* A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**
* T overcome this problem, the processes be **synchronized** in some way

##### *Solution to* Critical-Section Problem

* System consisting of n processes {P0, P1 , ... , Pn-1}, each process has a segment of code, called a critical section
* Feature: when one process is executing in its critical section, no other process is to be allowed to execute in its critical section
* The critical-section problem is to design a protocol that the processes can use to cooperate.



//request to critical section

//remaining code  
**Figure: General structure of a typical process Pi**

1. Mutual Exclusion - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting – There exists a bound or limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

* Assume that each process executes at a nonzero speed
* However, no assumption concerning **relative speed** of the n processes
* Two general approaches are used to handle critical sections in operating systems
  + A **preemptive kernel** allows a process to be preempted while it is running in kernel mode.
  + A **nonpreemptive kernel** does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode

##### Peterson’s Solution

* A classic software-based solution to the critical-section problem known as Peterson's solution
* Two process solution
* Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted. The two processes share two variables:
* int turn;
* Boolean flag[2];

The variable turn indicates whose turn it is to enter the critical section. turn = j, the process Pj is allowed to execute, in its critical section.

The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = TRUE, implies that process Pi is ready!

##### Algorithm for Process Pi (Pi is ready)

##### 

##### 

##### Algorithm for Process Pj (Pj is ready)

do {

flag[j] = TRUE; turn = i;

while (flag[i] && turn == i); critical section

flag[j] = FALSE;

remainder section

} while (TRUE);

* The eventual value of turn determines which of the two processes is allowed to enter its critical section first.
* This solution is correct so, we need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

* Since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

##### Synchronization Hardware

* Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures
* Solution to the critical-section problem requires a simple tool-a **lock**
* Initial value of lock is FALSE

##### Solution to Critical-section Problem Using Locks

do {

acquire lock

critical section release lock

remainder section

} while (TRUE);

* Race conditions are prevented by requiring that critical regions be protected by locks.
* Many systems provide hardware support for critical section code to make easier and improve system efficiency
* The critical-section problem could be solved simply in a Uniprocessors – could disable interrupts
* Currently running code would execute without preemption
* Generally too inefficient on multiprocessor systems while disabling interrupts on a multiprocessor can be time consuming
* Modern machines provide special atomic hardware instructions [Atomic = non-interrupt table]
* That allow us either test and modify the content of a memory word and set value Or swap contents of two memory words **atomically**

**TestAndSet Instruction Definition:** (performing in single instruction)

boolean TestAndSet (boolean \*target)

{

boolean rv = \*target;

\*target = TRUE; return rv;

}

##### P0 => target/ lock(T/F) = false // lock is global variable

##### rv (temporary variable) = false

##### target = true

##### return false

**Solution:** **Mutual-exclusion implementation using TestAndSet:**

Shared boolean variable lock, initialized to false. Solution:

do {

while ( TestAndSet (&lock ))

; // do nothing

// critical section lock = FALSE;

// remainder section

} while (TRUE);

##### Now &lock=false, so P0 stops and waits for a chance

##### P1 => target/ lock(T/F) = ture

##### rv (temporary variable) = ture

##### target = true

##### return ture

##### Mutex(Done), Progress(Done), Bounded waiting(Not Done) why because P1 will do the process until it becomes false but it never becomes false since it is in while loop, so P2 is in waiting state

##### If process contains 4 units of time, then P1 process only takes the 4 unit of time, no other process going to do task.

##### Bounded-waiting Mutual Exclusion with TestandSet()

do {

waiting[i] = TRUE; key = TRUE;

while (waiting[i] && key)

key = TestAndSet(&lock); waiting[i] = FALSE;

// critical section j = (i + 1) % n;

while ((j != i) && !waiting[j]) j = (j + 1) % n;

if (j == i)

lock = FALSE;

else

waiting[j] = FALSE;

// remainder section

} while (TRUE);

##### Swap Instruction (performing in single instruction)

Definition:

void Swap (boolean \*a, boolean \*b)

{

boolean temp = \*a;

\*a = \*b;

\*b = temp:

}

##### Solution using Swap

Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key Solution:

do {

key = TRUE; // key local variable

while ( key == TRUE) Swap (&lock, &key );

// critical section lock = FALSE;

// remainder section

} while (TRUE);

But it suffers from bounding waiting (i.e) some Process may for more time

##### Semaphore

* The hardware-based solutions to the critical-section problem presented in Synchronization Hardware are complicated for application programmers to use.
* To overcome this difficulty, we can use a synchronization tool called a **semaphore**
* Synchronization tool that does not require busy waiting
* Semaphore *S* – integer variable
* Two standard atomic operations modify S: wait() and signal()
* The wait () operation was originally termed P and signal() operation was originally called V
* Less complicated

The definition of wait() is as follows:

wait (S) {

while S <= 0

; // no-op

S--;

}

The definition of signal() is as follows:

signal (S) {

S++;

}

##### Semaphore as General Synchronization Tool/Usage:

* Counting semaphore – integer value can range over an unrestricted domain
* Binary semaphore – integer value can range only between 0 and 1; Also known as **mutex locks** and can implement a counting semaphore S as a binary semaphore
* Provides mutual exclusion

Semaphore mutex; // initialized to 1

do {

wait (mutex);

// Critical Section

signal (mutex);

// remainder section

} while (TRUE);

* The counting semaphore is initialized to the number of resources available
* Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
* When a process releases a resource, it performs a signal() operation (incrementing the count).
* When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.
* We can also use semaphores to solve various synchronization problems:

P1 execute statement S1 and P2 execute statement S2

S1;

signal(synch) ;

wait(synch);

S2;

Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 has been executed.

##### Semaphore Implementation:

 Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time

 Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.

 The disadvantage of the semaphore definition is **busy waiting** in critical section implementation

* Note that applications may spend lots of time in critical sections and therefore this is not a good solution.
* Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spin lock** because the process "spins" while waiting for the lock.

##### Semaphore Implementation with no Busy waiting

 With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:

value (of type integer)

pointer to next record in the list

To implement semaphores under this definition, we define a semaphore as a "C' struct:

typedef struct {

int value;

struct process \*list;

} semaphore;

* Two operations:

block – suspends the process that invokes it/ place the process invoking the operation on the appropriate waiting queue.

wakeup – The wakeup(P) operation resumes the execution of a blocked process P /remove one of processes in the waiting queue and place it in the ready queue

.

Implementation of wait:

wait(semaphore \*S) {

S->value--;

if (S->value < 0) {

add this process to S->list; block();

}

}

Implementation of signal:

signal(semaphore \*S) {

S->value++;

if (S->value <= 0) {

remove a process P from S->list; wakeup(P);

}

}

* The list of waiting processes can be easily implemented by a link field in each process control block (PCB).
* Each semaphore contains an integer value and a pointer to a list of PCBs.
* One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue

##### Deadlock and Starvation:

* Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes. The event in question is the execution of a signal() operation
* Let Po and P1, each accessing two semaphores, S and Q initialized to 1

|  |  |  |
| --- | --- | --- |
| *P*0  wait (S);  wait (Q);  . | *P*1  wait (Q);  wait (S); |  |
| .  signal (S); | signal (Q); | .  . |
| signal (Q); | signal (S); |  |

* Suppose that Po executes wait (S) and then P1 executes wait (Q). When Po executes wait (Q), it must wait until P1 executes signal (Q).
* Starvation/indefinite blocking – A process may never be removed from the semaphore queue in which it is suspended.(suspended -temporary blocking)

**Priority Inversion:**

* Scheduling problem when lower-priority process holds a lock needed by higher- priority process
* These systems solve the problem by implementing a **priority- inheritance protocol**
* **Example: L<M<H**
* A priority-inheritance protocol would allow process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution.
* When process L had finished using resource R, it would relinquish its inherited priority from H and assume its original priority.
* Because resource R would now be available, process H-not M-would run next.

**Classical Problems of Synchronization** Bounded-Buffer Problem Readers and Writers Problem Dining-Philosophers Problem

##### Bounded-Buffer Problem

*N* buffers, each can hold one item Semaphore mutex initialized to the value 1 Semaphore full initialized to the value 0 Semaphore empty initialized to the value N.

The structure of the producer process

do {

…

// produce an item in nextp

…

wait (empty);

wait (mutex);

…

// add the item to the buffer

…

signal (mutex);

signal (full);

} while (TRUE);

The structure of the consumer process do {

wait (full);

wait (mutex);

…

// remove an item from buffer to nextc

….

signal (mutex);

signal (empty);

…

// consume the item in nextc

…

} while (TRUE);

##### Readers-Writers Problem:

A data set is shared among a number of concurrent processes

* Readers – only read the data set; they do not perform any updates
* Writers – can both read and written
* Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
* First readers-writers problem, no reader should wait for other readers to finish simply because a writer is waiting
* Second readers-writers problem, if a writer is waiting to access the object, no new readers may start reading
* The reader processes share the following data structures:

semaphore mutex, wrt;

int readcount;

* Data set

Semaphore mutex initialized to 1 Semaphore wrt initialized to 1 Integer readcount initialized to 0

The structure of a writer process

do {

wait (wrt) ;

// writing is performed signal (wrt) ;

} while (TRUE);

The structure of a reader process do {

wait (mutex) ; readcount ++ ;

if (readcount == 1)

wait (wrt) ; signal (mutex)

// reading is performed wait (mutex) ;

readcount - - ;

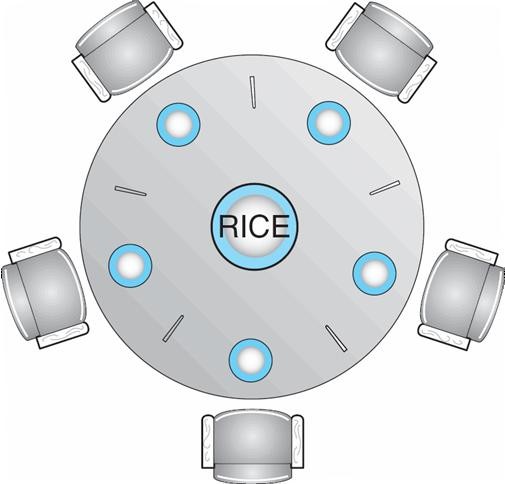
if (readcount == 0)

signal (wrt) ; signal (mutex) ;

} while (TRUE);

* Reader-writer locks are most useful in the following situations:
* In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
* In applications that have more readers than writers. This is because reader-writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader-writer lock.

##### Dining-Philosophers Problem:

* Shared data
* Bowl of rice (data set)
* The dining-philosophers problem is considered a classic synchronization problem
* It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
* Semaphore chopstick [5];
* Chopstick are initialized to 1
* The structure of Philosopher i:

do {

wait ( chopstick[i] );

wait ( chopStick[ (i + 1) % 5] );

// eat

signal ( chopstick[i] );

signal (chopstick[ (i + 1) % 5] );

// think

} while (TRUE);

* Several possible remedies to the deadlock problem are listed next.

##### Allow at most four philosophers to be sitting simultaneously at the table.

* Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
* Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick

##### Problems with Semaphores

Incorrect use of semaphore operations:

signal (mutex) …. wait (mutex)

wait (mutex) … wait (mutex)

Omitting of wait (mutex) or signal (mutex) (or both). In this case, either mutual exclusion is violated or a deadlock will

occur.

##### Monitors

* A high-level abstraction that provides a convenient and effective mechanism for process synchronization
* Only one process may be active within the monitor at a time

**Usage:**

**Syntax of monitor**

monitor monitor-name

{

// shared variable declarations procedure P1 (…) { …. }

…

procedure Pn (…) {……} Initialization code ( ….) { … }

…

}

##### Schematic view of a Monitor

**Condition Variables(additional synchronization scheme/tailor made synchronization)**

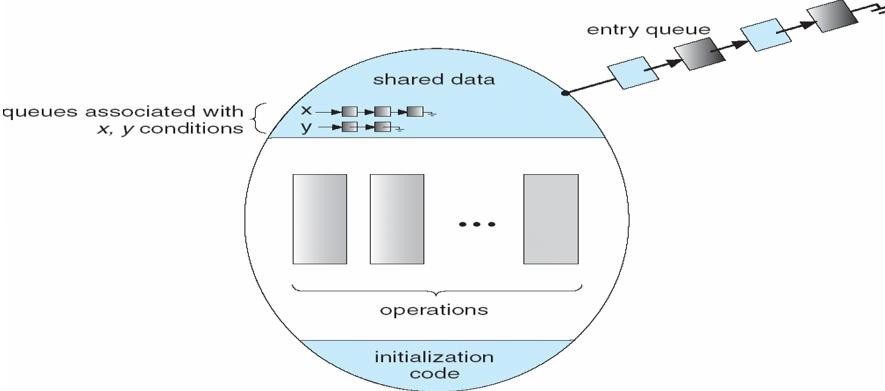
condition x, y;

Two operations on a condition variable:

x.wait () – a process that invokes the operation is suspended.

x.signal () – resumes one of processes (if any) that invoked x.wait ()

##### Monitor with Condition Variables





Now suppose that, when the x. signal () operation is invoked by a process P, there exists a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. Both processes P and Q can conceptually continue with their execution.

Two possibilities exist:

**Signal and wait.** *P* either waits until *Q* leaves the monitor or waits for another condition.

**Signal and continue.** *Q* either waits until *P* leaves the monitor or waits for another condition.

On the one hand, since *P* was already executing in the monitor, the *signal-and-continue* method seems more reasonable.

**Dining-Philosophers Solution Using Monitors:**

Each philosopher *i* invokes the operations pickup() and putdown() in the following sequence:

DiningPhilosophters.pickup (i); eat

DiningPhilosophers.putdown (i);

monitor dp

{

enum { THINKING; HUNGRY, EATING} state [5] ; //5 persons in any of these 3 states

condition self [5]; //Pi can delay herself when she is hungry but is unable to obtain the chopsticks she needs

void pickup (int i) {

state[i] = HUNGRY;

test(i); // it checks whether left and right persons are EATING

if (state[i] != EATING)

self [i].wait;

}

void putdown (int i) {

state[i] = THINKING;

// test left and right neighbors test((i + 4) % 5);

test((i + 1) % 5);

}

void test (int i) {

if ( (state[(i + 4) % 5] != EATING) &&

(state[i] == HUNGRY) &&

(state[(i + 1) % 5] != EATING) ) {

state[i] = EATING ;

self[i].signal () ;

}

}

initialization\_code() {

for (int i = 0; i < 5; i++)

state[i] = THINKING;

}

}

##### Monitor Implementation Using Semaphores:

**Variables**

semaphore mutex; // (initially = 1)

semaphore next; // (initially = 0) suspend themselves

int next\_count = 0; //to count the number of processes suspended on next

Each procedure ***F*** will be replaced by

wait(mutex);

…

body of *F*;

…

if (next\_count > 0) signal(next)

else

signal(mutex);

Mutual exclusion within a monitor is ensured.

##### Monitor Implementation:

For each condition variable ***x***, we have:

semaphore x\_sem; // (initially = 0)

int x\_count = 0;

The operation x.wait can be implemented as:

x\_count++;

if (next\_count > 0) signal(next);

else

signal(mutex);

wait(x\_sem);

x\_count--;

The operation x.signal can be implemented as:

if (x\_count > 0) {

next\_count++; signal(x\_sem); wait(next); next\_count--;

}

##### Resuming Processes within a Monitor:

monitor ResourceAllocator

{

boolean busy; condition x;

void acquire(int time) {

if (busy)

x.wait(time);

busy = TRUE;

}

void release() {

busy = FALSE; x.signal();

}

}

initialization code() {

busy = FALSE;

}

* In many circumstances, however, such a simple scheduling scheme is not adequate. For this purpose, the **conditional wait** construct can be used; it has the form
* x.wait(c);
* where c is an integer expression that is evaluated when the wait () operation is executed. The value of c, which is called a **priority number** is then stored with the name of the process that is suspended. When x. signal () is executed, the process with the smallest priority number is resumed next.
* A process that needs to access the resource in question must observe the following sequence:

R.acquire(t);

…

access the resource;

…

R. release() ;

where R is an instance of type ResourceAllocator.

* In particular, the following problems can occur:
  + A process might access a resource without first gaining access permission to the resource.
  + A process might never release a resource once it has been granted access to the resource.
  + A process might attempt to release a resource that it never requested.
  + A process might request the same resource twice (without first releasing the resource).

##### Synchronization Examples

Solaris Windows XP Linux Pthreads

##### Solaris Synchronization:

 Implements a variety of locks to supportmultitasking, multithreading (including real-time threads), and multiprocessing

Uses adaptive mutexes for efficiency, when protecting data from short code segments

Uses condition variables and readers-writers locks, when longer sections of code need access to data

Uses turnstiles, to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock. A **turnstile** is a queue structure containing threads blocked on a lock. Turnstiles are organized according to a **priority inheritance protocol**

##### Windows XP Synchronization:

Uses interrupt masks, to protect access to global resources on uniprocessor systems Uses spinlocks on multiprocessor systems

Also provides **dispatcher objects** which may act as either mutexes and semaphores Dispatcher objects may also provide events

Dispatcher objects may be in either a signaled state or a nonsignaled state



* An event acts much like a condition variable

##### Linux Synchronization:

* Linux: Prior to kernel Version 2.6, it is a nonpreemptive kernel
* Version 2.6 and later, fully preemptive
* Linux provides: semaphores and spin locks



##### Pthreads Synchronization:

* Pthreads API is OS-independent
* It provides: mutex locks, condition variables and read-write locks
* Other extensions to the Pthreads API include spinlocks, but not all extensions are considered portable from one implementation to another

##### Atomic Transactions

* Assures that operations happen as a single logical unit of work, in its entirety, or not at all
* Related to field of database systems
* Challenge is assuring atomicity despite computer system failures
* Example: fund transfer

##### System Model:

* + **Transaction** - collection of instructions or operations that performs single logical function
  + Here we are concerned with changes to stable storage – disk
  + Transaction is a sequence of read and write operations
  + Terminated by commit (transaction successful) or abort (transaction failed) operation. Aborted transaction must be rolled back to undo any changes it performed

##### Types of Storage Media

* **Volatile storage** – information stored here does not survive system crashes (Example: main memory, cache)
* **Nonvolatile storage** – Information usually survives crashes( Example: disk and tape)
* **Stable storage** – Information never lost
* Not actually possible, so approximated via replication or RAID to devices with independent failure modes
* Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

##### Log-Based Recovery:

Record to stable storage information about all modifications by a transaction The most widely used method is **write-ahead logging**

Log on stable storage, each **log** record describes single transaction write operation, including

**Transaction name:** The unique name of the transaction that performed the write operation

**Data item name:** The unique name of the data item written

**Old value:** The value of the data item prior to the write operation

**New value:** The value that the data item will have after the write

<Ti starts> written to log when transaction Ti starts

<Ti commits> written when Ti commits

Log entry must reach stable storage before operation on data occurs

##### Log-Based Recovery Algorithm

Using the log, system can handle any volatile memory errors

Undo(Ti) restores value of all data updated by Ti to old values

Redo(Ti) sets values of all data in transaction Ti to new values Undo(Ti) and redo(Ti) must be idempotent



Idempotent-Multiple executions must have the same result as one execution

If system fails, restore state of all updated data via log

* If log contains <Ti starts> without <Ti commits>, undo(Ti)
* If log contains <Ti starts> and <Ti commits>, redo(Ti)

##### Checkpoints:

* Log could become long when we determine which transactions need to be redone and which need to be undone, and recovery could take long
* Checkpoints shorten log and recovery time.
* Checkpoint scheme:

1. Output all log records currently in volatile storage to stable storage

2. Output all modified data residing in volatile to stable storage

3. Output a log record <checkpoint> onto stable storage

* Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti. All other transactions already on stable storage
* The recovery operations that are required are as follows:
* For all transactions Tk in *T* for which the record < Tk commits> appears in the log, execute redo (Tk)·
* For all transactions Tk in *T* that have no < Tk commits> record in the log, execute undo (Tk).

##### Concurrent Atomic Transactions:

* Each transaction is atomic, the concurrent execution of transactions must be equivalent to the case where these transactions are executed serially in some arbitrary order – serializability
* Could perform all transactions in critical section
* All transactions share a common semaphore mutex, which is initialized to 1. When a transaction starts executing, its first action is to execute wait(mutex). After the transaction either commits or aborts, it executes signal(mutex)
* Inefficient, too restrictive
* Concurrency-control algorithms provide serializability

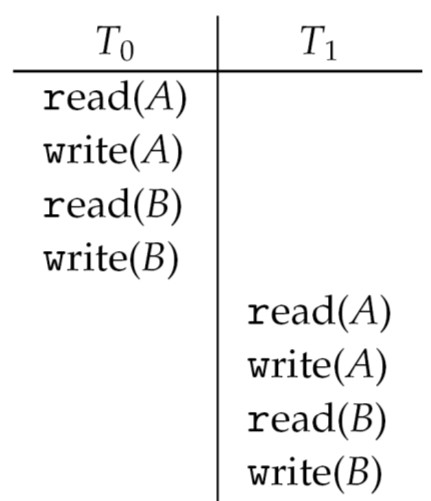
##### Serializability

 Consider two data items A and B Consider Transactions T0 and T1 Execute T0, T1 atomically Execution sequence called schedule

Atomically executed transaction order called serial schedule

For n transactions, there exist n! different valid serial schedules

##### Schedule 1: T0 then T1



**Nonserial Schedule**

* Nonserial schedule allows overlapped execution and it does not necessarily imply an incorrect execution
* Consider schedule S, operations Oi, Oj of transactions Ti and Tj
* Oi, Oj conflict if access same data item, with at least one write operation

##### Schedule 2: Concurrent Serializable Schedule

The write(A) operation of *To* conflicts with the read(A) operation of T1. However, the write(A) operation of T1 does not conflict with the read(B) operation of *To,* because the two operations access different data items.

* Let Oi and Oj be consecutive operations of a schedule S.
* If Oi and Oj are operations of different transactions and Oi and Oj do not conflict then we can swap the order of Oi and Oj to produce a new schedule S'.
* Continuing with this procedure of swapping nonconflicting operations, we get:
* Swap the read(B) operation of To with the read(A) operation of T1.
  + Swap the write(B) operation of To with the write(A) operation of T1.
* S is conflict serializable

**Locking Protocol:**

* To ensure serializability by associating **lock** with each data item
* Each transaction follow a **locking protocol** that governs how locks are acquired and released
* Two models:
  + **Shared** – Ti has shared-mode lock (S) on data item Q, Ti can read Q but cannot write Q
  + **Exclusive** – Ti has exclusive-mode lock (X) on Q, Ti can read and write Q
* Require every transaction on data item Q to acquire appropriate lock
* If lock already held, new request may have to wait
* It is similar to readers-writers algorithm

##### Two-phase Locking Protocol

It generally ensures conflict serializability

Each transaction issues lock and unlock requests in two phases

**Growing** –A transaction may obtain locks but may not release any lock

**Shrinking** – A transaction may release locks but may not obtain any new locks

The two-phase locking protocol ensures conflict serializability. It does not prevent deadlock

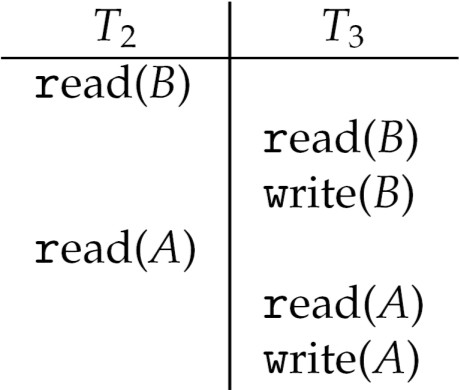
##### Timestamp-based Protocols:

* In the locking protocols described above, the order followed by pairs of conflicting transactions is determined at execution time
* Another method is **Timestamp-ordering :** Select an order among transactions in advance
* Transaction Ti associated with timestamp TS(Ti) before Ti starts
* TS(Ti) < TS(Tj) if Ti entered system before Tj
* TS can be generated from system clock or as logical counter incremented at each entry of transaction
* Timestamps determine serializability order
* If TS(Ti) < TS(Tj), system must ensure produced schedule equivalent to serial schedule where Ti appears before Tj

##### Timestamp-based Protocol Implementation

* Data item Q gets two timestamps
* **W-timestamp**(Q) – largest timestamp of any transaction that successfully executed write(Q)
* **R-timestamp**(Q) – largest timestamp of any transaction that successful executed read(Q)
* The Timestamps are updated whenever read(Q) or write(Q) executed
* Timestamp-ordering protocol ensures any conflicting read and write executed in timestamp order
* Suppose Ti executes read(Q)
* If TS(Ti) < W-timestamp(Q), Ti needs to read value of Q that was already overwritten read operation rejected and Ti rolled back
* If TS(Ti) ≥ W-timestamp(Q) read executed, R-timestamp(Q) set to max(R-timestamp(Q), TS(Ti))
* Suppose Ti executes write (Q)
* If TS(Ti) < R-timestamp(Q), value Q produced by Ti was needed previously and Ti assumed it would never be produced Write operation rejected, Ti rolled back
* If TS(Ti) < W-timestamp(Q), Ti attempting to write obsolete value of Q Write operation rejected and Ti rolled back
* Otherwise, write executed
* Any rolled back transaction Ti is assigned a new timestamp and restarted

**Schedule Possible Under Timestamp Protocol**



* Algorithm ensures conflict serializability and freedom from deadlock