Understanding Operating Systems
Sixth Edition

Chapter 6
Concurrent Processes

Learning Objectives

2

After completing this chapter, you should be able to describe:

- The critical difference between processes and processors, and their connection
- The differences among common configurations of multiprocessing systems
- The significance of a critical region in process synchronization
- The basic concepts of process synchronization software: test-and-set, WAIT and SIGNAL, and semaphores

Learning Objectives (cont'd.)

3

- The need for process cooperation when several processes work together
- How several processors, executing a single job, cooperate
- The similarities and differences between processes and threads
- The significance of concurrent programming languages and their applications

What Is Parallel Processing?

4

- Parallel processing
  - Two or more processors operate in one system at the same time
    - Work may or may not be related
  - Two or more CPUs execute instructions simultaneously
  - Processor Manager
    - Coordinates activity of each processor
    - Synchronizes interaction among CPUs

Parallel Processing Benefits

5

- Parallel processing development
  - Enhances throughput
  - Increases computing power
- Benefits
  - Increased reliability
    - More than one CPU
    - If one processor fails, others take over
  - Faster processing
    - Instructions processed in parallel two or more at a time
- Not simple to implement

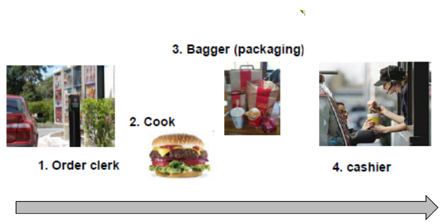
Parallel Processing? (cont'd.)

6

- Faster instruction processing methods
  - CPU allocated to each program or job
  - CPU allocated to each working set or parts of it
  - Individual instructions subdivided
    - Each subdivision processed simultaneously
    - **Concurrent programming**
- Two major challenges
  - Connecting processors into configurations
  - Orchestrating processor interaction

Example: Drive-Through

7



What Is Parallel Processing? (cont'd.)

8

(table 6.1)

The six steps of the fast-food lunch stop.

Originator	Action	Receiver
Processor 1 (the order clerk)	Accepts the query, checks for errors, and passes the request on to =>	Processor 2 (the bagger)
Processor 2 (the bagger)	Searches the database for the required information (the hamburger)	
Processor 3 (the cook)	Retrieves the data from the database (the meat to cook for the hamburger) if it's kept off-line in secondary storage	
Processor 3 (the cook)	Once the data is gathered (the hamburger is cooked), it's placed where the receiver => can get it (in the hamburger bin)	Processor 2 (the bagger)
Processor 2 (the bagger)	Passes it on to =>	Processor 4 (the cashier)
Processor 4 (the cashier)	Routes the response (your order) back to the originator of the request =>	You

Levels of Multiprocessing

9

- Developed for high-end midrange and mainframe computers
  - Each additional CPU treated as additional resource
- Today hardware costs reduced
  - Multiprocessor systems available on all systems
- Multiprocessing occurs at three levels
  - Job level
  - Process level
  - Thread level
    - Each requires different synchronization frequency

Levels of Multiprocessing (cont'd.)

Parallelism Level	Process Assignments	Synchronization Required
Job Level	Each job has its own processor, and all processes and threads are run by that same processor.	No explicit synchronization required after jobs are assigned to a processor.
Process Level	Unrelated processes, regardless of job, can be assigned to available processors.	Moderate amount of synchronization required.
Thread Level	Threads, regardless of job and process, can be assigned to available processors.	High degree of synchronization required to track each process and each thread.

(table 6.2)
Typical levels of parallelism and the required synchronization among processors.

Introduction to Multi-Core Processors

11

- Multi-core processing
  - Several processors placed on single chip
- Problems
  - Heat and current leakage (tunneling)
- Solution
  - Single chip with two processor cores in same space
    - Allows two sets of simultaneous calculations
    - Thousands of cores on single chip
  - Two cores each run more slowly than single core chip

Typical Multiprocessing Configurations

12

- Multiple processor configuration impacts systems
- Three types
  - Master/slave
  - Loosely coupled
  - Symmetric

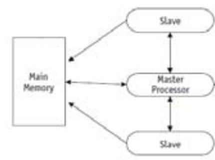
Master/Slave Configuration

13

- Asymmetric multiprocessing system
- Single-processor system
  - Additional slave processors
    - Each managed by primary master processor
- Master processor responsibilities
  - Manages entire system
  - Maintains all processor status
  - Performs storage management activities
  - Schedules work for other processors
  - Executes all control programs

Master/Slave Configuration (cont'd.)

14

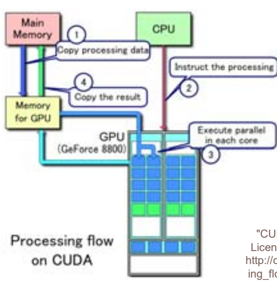


(figure 6.1)

In a master/slave multiprocessing configuration, slave processors can access main memory directly but they must send all I/O requests through the master processor.

CUDA CORES

15



- Java – jCUDA, JCuda, JCublas, JCuf
  CUDA4J
- MATLAB – Parallel Computing Toolbo
  MATLAB Distributed Computing
  Server,[23] and 3rd party packages lik
  Jacket.
- Perl – KappaCUDA, CUDA::Minimal
- Python – Numba, NumbaPro, PyCUD
  KappaCUDA, Theano
- Ruby – KappaCUDA
- R – gputools

Features

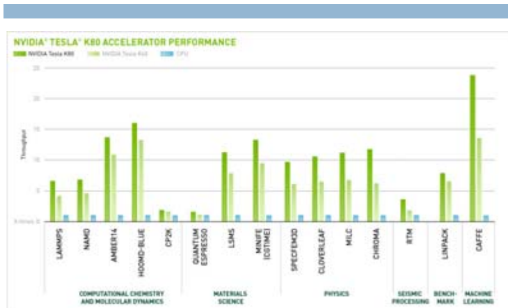
16

Features	Tesla K80[1]	Tesla K40
GPU	2x Kepler GK210	1 Kepler GK110B
Peak double precision floating point performance	2.91 Tflops (GPU Boost Clocks) 1.87 Tflops (Base Clocks)	1.66 Tflops (GPU Boost Clocks) 1.43 Tflops (Base Clocks)
Peak single precision floating point performance	8.74 Tflops (GPU Boost Clocks) 5.6 Tflops (Base Clocks)	5 Tflops (GPU Boost Clocks) 4.29 Tflops (Base Clocks)
Memory bandwidth (ECC off)[2]	480 GB/sec (240 GB/sec per GPU)	288 GB/sec
Memory size (GDDR5)	24 GB (12GB per GPU)	12 GB
CUDA cores	4992 (2496 per GPU)	2880

- See more at: http://www.nvidia.com/object/tesla-servers.html#sthash.rOdwBlmr.dpuf

Where is it being used?

17



Master/Slave Configuration (cont'd.)

18

- Advantages
  - Simplicity
- Disadvantages
  - Reliability
    - No higher than single processor system (If the master fails all system fails)
  - Potentially poor resources usage
    - If master is busy it cannot assign job to the slaves
  - Increases number of interrupts
    - All slaves must interrupt the master for I/O operations.

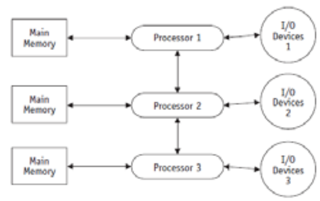
Loosely Coupled Configuration

19

- Several complete computer systems
  - Each with own resources
    - Memory, I/O, CPU, operating system
  - Each processor
    - Communicates and cooperates with others
- Several requirements and policies for job scheduling
- Single processor failure
  - Others continue work independently
  - Difficult to detect

Loosely Coupled Configuration (cont'd.)

20



(figure 6.2)
In a loosely coupled multiprocessing configuration, each processor has its own dedicated resources.

Symmetric Configuration

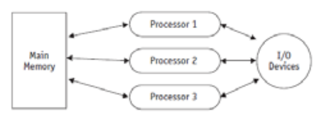
21

- Decentralized processor scheduling
  - Each processor is same type
- Advantages (over loosely coupled configuration)
  - More reliable
  - Uses resources effectively
  - Can balance loads well
  - Can degrade gracefully in failure situation
- Most difficult to implement
  - Requires well synchronized processes
    - Avoids races and deadlocks

Symmetric Configuration (cont'd.)

22

(figure 6.3)
A symmetric multiprocessing configuration with homogeneous processors. Processes must be carefully synchronized to avoid deadlocks and starvation.



Symmetric Configuration (cont'd.)

23

- Decentralized processor scheduling
  - Single operating system copy
  - Global table listing
- More conflicts
  - Several processors access same resource at same time
- **Process synchronization**
  - Algorithms resolving conflicts between processors
  - Requires Process Synchronization Software

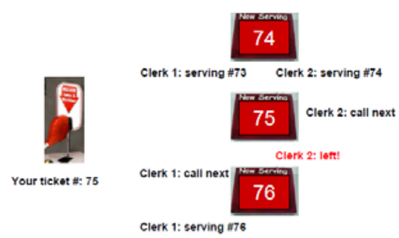
Process Synchronization Software

24

- Resolving conflicts between processors
- Successful process synchronization
  - Lock up used resource
    - Protect from other processes until released
  - Only when resource is released
    - Waiting process is allowed to use resource
- Mistakes in synchronization can result in:
  - Starvation
  - Deadlock

An unpleasant scenario

25



Process Synchronization Software (cont'd.)

26

- **Critical region**
  - Part of a program
  - Critical region must complete execution
    - Other processes must wait before accessing critical region resources
    - Example: clerk must wait the customer to show up, cannot take other calls.
- Processes within critical region
  - Cannot be interleaved
    - Threatens integrity of operation

Process Synchronization Software (cont'd.)

27

- Synchronization
  - Implemented as lock-and-key arrangement:
  - Process determines key availability
    - Process obtains key
    - Puts key in lock
    - Makes it unavailable to other processes
- Types of locking mechanisms
  - Test-and-set
  - WAIT and SIGNAL
  - Semaphores

Four Criteria to solve critical section problem

28

1. No two processes may be simultaneously into their critical sections for the same shared data

   We want to enforce mutual exclusion

2. No assumptions should be made about the speeds at which the processes will execute.

   The solution should be general:
   the actual speeds at which processes complete are often impossible to predict because running processes can be interrupted at any time!

Four Criteria to solve critical section problem

29

3. No process should be prevented to enter its critical section when no other process is inside its own critical section for the same shared data

   Should not prevent access to the shared data when it is not necessary

4. No process should have to wait forever to enter a critical section

   Solution should not cause starvation

.

Peterson's algorithm

30

```
#define LOCKED 1
#define UNLOCKED 0
int lock = UNLOCKED; // shared
```

```
// Start busy wait
while (lock == LOCKED);
lock = LOCKED;
   Critical_Section();
//Leave critical section
lock = UNLOCKED;
```

Failure

31

- Solution fails if two or more processes reach the critical section ***in lockstep*** when the critical section is UNLOCKED
  - Will both exit the busy wait
  - Will both set Lock variable to LOCKED
  - Will both enter the critical section at the same time
- ***Which condition does this solution violate?***
- ***Solution violates second condition***
  - Does not ***always*** guarantee mutual exclusion

Second bad solution

32

- We have ***two processes***
  - Their process ID's are **0** and **1**

```
int turn=0;  // shared

void enter_region(int pid){ /* Start busy
wait*/  while (turn != pid); }
//enter_region

void leave_region(int pid){ turn = 1 –
pid  /* 1 if 0 and 0 if 1 } */
leave_region
```


Failure 2

33

- Solution works well as long as the two processes take turns
  - If one process misses its turn, other process cannot enter the critical section
  - Not acceptable
- ***Which condition does this solution violate?***
- *Solution violates third condition*
  - *It prevents a process to enter its critical section at times when the other process is not inside its own critical section for the same shared data*

Another bad solution

34

- **Reserve first and check later**

```
#define F 0
#define T 1
// shared array
int reserved[2] = {F, F};
void enter_region(int pid) {
       int other; // pid of other
       other = 1 - pid;
       reserved[pid] = T;
       // Start busy wait
       while (reserved[other]);
 } // enter_region
void leave_region(int pid) {
       reserved[pid] = F;
  } // leave_region
```

35

- What if two processes try to enter their critical section at the same time and execute

```
reserved[pid] = T;
while (reserved[other]);
```

in lockstep?

- Will set both elements of `reserved` array to **T** and cause a **deadlock**
  - **Processes prevent each other from entering the critical section**
- ***Which condition does this solution violate?***

36

- ***Solution violates fourth condition***
  - *It causes a deadlock*

37

- Based on last bad solution but introduces a ***tiebreaker***

```
#define F 0
#define T 1
// shared variables
int reserved[2] = {F, F};
int mustwait; // tiebreaker

void enter_region(int pid) {
     int other; //other process
     other = 1 - pid;       reserved[pid] = T;
     // set tiebraker
     must_wait = pid;
     while ( reserved[other]&& must_wait==pid);
} // enter_region

void leave_region(int pid) {
     reserved[pid] = F;
} // leave_region
```

38

- Essential part of algorithm is

```
reserved[pid] = T;
must_wait = pid;
while reserved[other]&&
must_wait==pid);
```

- When two processes arrive ***in lockstep, last one must wait***

Test-and-Set

39

- Indivisible machine instruction
- Executed in single machine cycle
  - If key available: set to unavailable
- Actual key
  - Single bit in storage location: zero (free) or one (busy)
- Before process enters critical region
  - Tests condition code using TS instruction
  - No other process in region
    - Process proceeds
    - Condition code changed from zero to one
    - P1 exits: code reset to zero, allowing others to enter

Test-and-Set (cont'd.)

40

- Advantages
  - Simple procedure to implement
  - Works well for small number of processes
- Drawbacks
  - Starvation
    - Many processes waiting to enter a critical region
    - Processes gain access in arbitrary fashion
  - Busy waiting
    - Waiting processes remain in unproductive, resource-consuming wait loops

Why busy wait is bad

41

- **Busy waits waste CPU cycles:**
  - ***Generate unnecessary context switches***
  - ***Slow down the progress of other processes***
- A high priority process doing a busy wait may prevent a lower priority process to do its work and leave its critical region.
- *Think about a difficult boss calling you every two or three minutes to ask you about the status of the report you are working on*

Solution

42

- Several operating systems for multiprocessor architectures offer two different mutual exclusion mechanisms:
  - ***Busy waits*** for very ***short waits***
  - Putting the waiting process in the ***waiting state*** until the resource becomes free for longer waits

WAIT and SIGNAL

43

- Modification of test-and-set
  - Designed to remove busy waiting
- Two new mutually exclusive operations
  - WAIT (V) and SIGNAL (P)
  - Part of process scheduler's operations
- WAIT
  - Activated when process encounters busy condition code
- SIGNAL
  - Activated when process exits critical region and condition code set to "free"

Semaphores

44

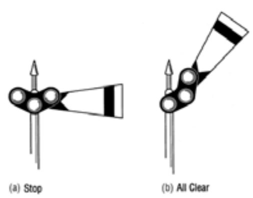
- Nonnegative integer variable
  - Flag
  - Signals if and when resource is free
    - Resource can be used by a process
- Two operations of semaphore
  - P (proberen means "to test")
  - V (verhogen means "to increment")

Semaphores (cont'd.)

45

(figure 6.4)

The semaphore used by railroads indicates whether your train can proceed. When it's lowered (a), another train is approaching and your train must stop to wait for it to pass. If it is raised (b), your train can continue.



Semaphores (cont'd.)

46

- Let s be a semaphore variable
  - V(s): $s := s + 1$
    - Fetch, increment, store sequence
  - P(s): If $s > 0$, then $s := s - 1$
    - Test, fetch, decrement, store sequence
- $s = 0$ implies busy critical region
  - Process calling on P operation must wait until $s > 0$
- Waiting job of choice processed next
  - Depends on process scheduler algorithm

The P( ) operation

47

- The P( ) operation
  - If semaphore value is zero,
    - Wait until value become positive
  - Once value of semaphore is greater than zero,
    - Decrement it
- The V() operation
  - Increment the value of the semaphore

How they work

48

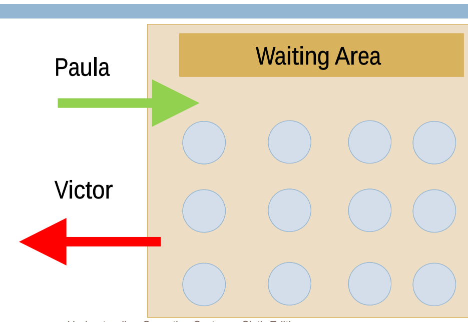
- The normal implementation of semaphores is through *system calls*:
  - ***Busy waits*** are ***eliminated***
  - Processes waiting for a semaphore whose value is zero are put in the ***waiting state***

An analogy

49

- Paula and Victor work in a restaurant:
- Paula handles customer arrivals:
  - Prevents people from entering the restaurant when all tables are busy.
- Victor handles departures
  - Notifies people waiting for a table when one becomes available
- The semaphore represents the number of available tables
  - Initialized with the ***total number of tables*** in restaurant

50



Semaphores (cont'd.)

51

State Number	Actions Calling Process	Operation	Running in Critical Region	Results Blocked on s	Value of s
0					1
1	P1	test(s)	P1		0
2	P1	increment(s)			1
3	P2	test(s)	P2		0
4	P3	test(s)	P2	P3	0
5	P4	test(s)	P2	P3, P4	0
6	P2	increment(s)	P3	P4	0
7			P3	P4	0
8	P3	increment(s)	P4		0
9	P4	increment(s)			1

(table 6.3)

The sequence of states for four processes calling test and increment (P and V) operations on the binary semaphore s. (Note: The value of the semaphore before the operation is shown on the line preceding the operation. The current value is on the same line.)

Semaphores (cont'd.)

52

- P and V operations on semaphore *s*
  - Enforce mutual exclusion concept
- Semaphore called **mutex** (MUTual EXclusion)
  - P(mutex): if mutex > 0 then mutex: = mutex – 1
  - V(mutex): mutex: = mutex + 1
- **Critical region**
  - Ensures parallel processes modify shared data only while in critical region
- Parallel computations
  - Mutual exclusion explicitly stated and maintained

Process Cooperation

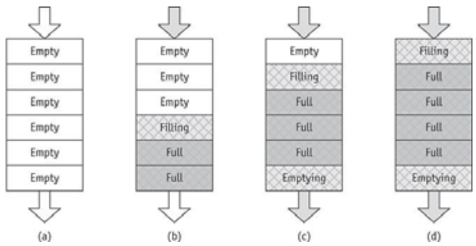
53

- Several processes work together to complete common task
- Each case requires
  - Mutual exclusion and synchronization
- Absence of mutual exclusion and synchronization
  - Results in problems
- Examples
  - Producers and consumers problem
  - Readers and writers problem
- Each case implemented using semaphores

Producers and Consumers

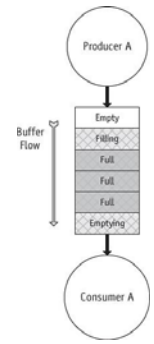
54

- One process produces data
- Another process later consumes data
- Example: CPU and line printer buffer
  - Delay producer: buffer full
  - Delay consumer: buffer empty
  - Implemented by two semaphores
    - Number of full positions
    - Number of empty positions
  - Mutex
    - Third semaphore: ensures mutual exclusion

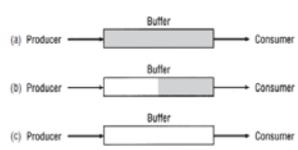


(figure 6.6)
Four snapshots of a single buffer in four states from completely empty (a) to almost full (d).
© Cengage Learning 2014

(figure 6.7)
Typical system with one producer, one consumer, and a single buffer.
© Cengage Learning 2014



Producers and Consumers (cont'd.)



(figure 6.5)
The buffer can be in any one of these three states: (a) full buffer, (b) partially empty buffer, or (c) empty buffer.

Producers and Consumers (cont'd.)

Producer	Consumer
produce data	P (full)
P (empty)	P (mutex)
P (mutex)	read data from buffer
write data into buffer	V (mutex)
V (mutex)	V (empty)
V (full)	consume data

- Producers and Consumers Algorithm

```
empty: = n
full: = 0
mutex: = 1
COBEGIN
  repeat PRODUCER until no more data
  repeat CONSUMER until buffer is empty
COEND
```



```
producer() {                    consumer() {
   struct x item;                  struct x item;
   for(;;) {                       for(;;) {
      produce(&item);                 P(&notempty);
      P(&notfull);                    P(&mutex);
      P(&mutex);                      take(item);
      put(item);                      V(&mutex);
      V(&mutex);                      V(&notfull);
      V(&notempty);                   eat(item);
   } // for                        } // for
} // producer                   } // consumer
```


Order matters

- The order of the two **P( )** operations is ***very important***
  - Neither the producer or the consumer should request exclusive access to the buffer before being sure they can perform the operation they have to perform
- The order of the two V( ) operations does not matter

Producers and Consumers (cont'd.)

61

(table 6.5) *Definitions of the elements in the Producers and Consumers Algorithm.*

Variables, Functions	Definitions
`full`	defined as a semaphore
`empty`	defined as a semaphore
`mutex`	defined as a semaphore
`n`	the maximum number of positions in the buffer
`V (x)`	$x := x + 1$ (x is any variable defined as a semaphore)
`P (x)`	if $x > 0$ then $x := x - 1$
`mutex = 1`	means the process is allowed to enter the critical region
`COBEGIN`	the delimiter that indicates the *beginning* of concurrent processing
`COEND`	the delimiter that indicates the *end* of concurrent processing

Producers and Consumers (cont'd.)

62

- Producers and Consumers Algorithm

```
empty: = n
full: = 0
mutex: = 1
COBEGIN
         repeat until no more data PRODUCER
         repeat until buffer is empty
  CONSUMER
COEND
```


Readers and Writers

63

- Two process types need to access shared resource
  - Example: file or database
- Example: airline reservation system
  - Implemented using two semaphores
    - Ensures mutual exclusion between readers and writers
  - Resource given to all readers
    - Provided no writers are processing (W2 = 0)
  - Resource given to a writer
    - Provided no readers are reading (R2 = 0) and no writers writing (W2 = 0)

Solution 1

64

Initialisation	Reader	Writer
mx = Semaphore(1) wrt = Semaphore(1) ctr = Integer(0)	- Wait mx - if (++ctr)==1, then Wait wrt - Signal mx [Critical section] - Wait mx - if (--ctr)==0, then Signal wrt - Signal mx	- Wait wrt [Critical section] - Signal wrt

Starvation of writer: if readers are continuously entering

"Faster Fair Solution for the Reader-Writer Problem", Popov et al. , 2013

Solution 2

65

Initialisation	Reader	Writer
in = Semaphore(1) mx = Semaphore(1) wrt = Semaphore(1) ctr = Integer(0)	- Wait in - Wait mx - if (++ctr)==1, then Wait wrt - Signal mx - Signal in [Critical section] - Wait mx - if (--ctr)==0, then Signal wrt - Signal mx	- Wait in - Wait wrt [Critical section] - Signal wrt - Signal in

Reader must lock two mutexes to enter to the critical area

"Faster Fair Solution for the Reader-Writer Problem", Popov et al. , 2013

Solution 3

66

Initialisation	Reader	Writer
in = Semaphore(1) out = Semaphore(1) wrt = Semaphore(0) ctrin = Integer(0) ctrout = Integer(0) wait = Boolean(0)	- Wait in - ctrin++ - Signal in [Critical section] - Wait out - ctrout++ - if (wait==1 && ctrin==ctrout) then Signal wrt - Signal out	- Wait in - Wait out - if (ctrin==ctrout) then Signal out else - wait=1 - Signal out - Wait wrt - wait=0 [Critical section] - Signal in

Faster solution since require only one mutex for reader to enter the critical region
If there are still readers in the critical region, writer waits until they leave but no more readers are allowed in

"Faster Fair Solution for the Reader-Writer Problem", Popov et al. , 2013

Concurrent Programming

67

- **Concurrent processing system**
  - One job uses several processors
    - Executes sets of instructions in parallel
  - Requires programming language and computer system support

Applications of Concurrent Programming

68

A = 3 * B * C + 4 / (D + E) ** (F – G)

(table 6.6)

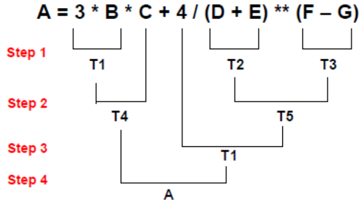
The sequential computation of the expression requires several steps. (In this example, there are seven steps, but each step may involve more than one machine operation.)

Step No.	Operation	Result
1	(F – G)	Store difference in T1
2	(D + E)	Store sum in T2
3	(T2) ** (T1)	Store power in T1
4	4 / (T1)	Store quotient in T2
5	3 * B	Store product in T1
6	(T1) * C	Store product in T1
7	(T1) + (T2)	Store sum in A

Applications of Concurrent Programming - cont

69

A = 3 * B * C + 4 / (D + E) ** (F – G)



Applications of Concurrent Programming (cont'd.)

70

A = 3 * B * C + 4 / (D + E) ** (F – G)

Step No.	Processor	Operation	Result
1	1	3 * B	Store product in T1
	2	(D + E)	Store sum in T2
	3	(F – G)	Store difference in T3
2	1	(T1) * C	Store product in T4
	2	(T2) ** (T3)	Store power in T5
3	1	4 / (T5)	Store quotient in T1
4	1	(T4) + (T1)	Store sum in A

(table 6.7)

With concurrent processing, the seven-step procedure can be processed in only four steps, which reduces execution time.

Applications of Concurrent Programming (cont'd.)

71

- **Explicit parallelism**
  - Requires programmer intervention
    - Explicitly state parallel executable instructions
  - Disadvantages
  - Time-consuming coding
  - Missed opportunities for parallel processing
  - Errors
    - Parallel processing mistakenly indicated
  - Programs difficult to modify

Applications of Concurrent Programming (cont'd.)

72

- **Implicit parallelism**
- Compiler automatically detects parallel instructions
- Advantages
  - Solves explicit parallelism problems
  - Complexity dramatically reduced
    - Working with array operations within loops
    - Performing matrix multiplication
    - Conducting parallel searches in databases
    - Sorting or merging file

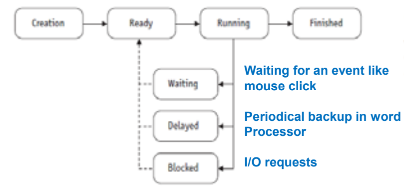
Threads and Concurrent Programming

73

- **Threads**
  - Small unit within process
    - Scheduled and executed
- Minimizes overhead
  - Swapping process between main memory and secondary storage
- Each active thread in a process has its own
  - Processor registers, program counter, stack and status
- Shares data area and resources allocated to its process

Thread States

74



(figure 6.6)
A typical thread changes states from READY to FINISHED as it moves through the system.

Thread States (cont'd.)

75

- Operating system support
  - Creating new threads
  - Setting up thread
    - Ready to execute
  - Delaying or putting threads to sleep
    - Specified amount of time
  - Blocking or suspending threads
    - Those waiting for I/O completion
  - Setting threads to WAIT state
    - Until specific event occurs

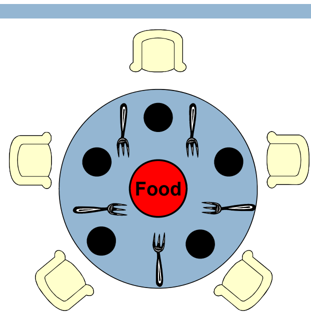
Thread States (cont'd.)

76

- Operating system support (cont'd.)
  - Scheduling thread execution
  - Synchronizing thread execution
    - Using semaphores, events, or conditional variables
  - Terminating thread
    - Releasing its resources

Revisiting Dining Philosophers

77



78

```
philosopher() {
   struct x spaghetti;
   for(;;) {

    think();

    eat(spaghetti);

  } // for
} // philosopher
```

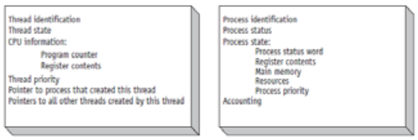
Thread Control Block

79

- Information about current status and characteristics of thread

(figure 6.7)

Comparison of a typical Thread Control Block (TCB) vs. a Process Control Block (PCB) from Chapter 4.



Concurrent Programming Languages

80

- **Ada**
  - First language providing specific concurrency commands
  - Developed in late 1970's
- **Java**
  - Designed as universal Internet application software platform
  - Developed by Sun Microsystems
  - Adopted in commercial and educational environments

Java

81

- Allows programmers to code applications that can run on any computer
- Developed at Sun Microsystems, Inc. (1995)
- Solves several issues
  - High software development costs for different incompatible computer architectures
  - Distributed client-server environment needs
  - Internet and World Wide Web growth
- Uses compiler and interpreter
  - Easy to distribute

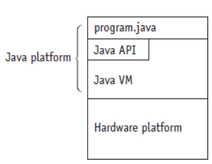
Java (cont'd.)

82

- **The Java Platform**
- Software only platform
  - Runs on top of other hardware-based platforms
- Two components
  - Java Virtual Machine (Java VM)
    - Foundation for Java platform
    - Contains the interpreter
    - Runs compiled bytecodes
  - Java application programming interface (Java API)
    - Collection of software modules
    - Grouped into libraries by classes and interfaces

Java (cont'd.)

83



(figure 6.8)

A process used by the Java platform to shield a Java program from a computer's hardware.

Java (cont'd.)

84

- **The Java Language Environment**
- Designed for experienced programmers (like C++)
- Object oriented
  - Exploits modern software development methods
    - Fits into distributed client-server applications
- Memory allocation features
  - Done at run time
  - References memory via symbolic "handles"
  - Translated to real memory addresses at run time
  - Not visible to programmers

Java (cont'd.)

85

- Security
  - Built-in feature
    - Language and run-time system
  - Checking
    - Compile-time and run-time
- Sophisticated synchronization capabilities
  - Multithreading at language level
- Popular features
  - Handles many applications; can write a program once; robust; Internet and Web integration

Summary

86

- Multiprocessing
  - Single-processor systems
    - Interacting processes obtain control of CPU at different times
  - Systems with two or more CPUs
    - Control synchronized by processor manager
    - Processor communication and cooperation
  - System configuration
    - Master/slave, loosely coupled, symmetric

Summary (cont'd.)

87

- Multiprocessing system success
  - Synchronization of resources
- Mutual exclusion
  - Prevents deadlock
  - Maintained with test-and-set, WAIT and SIGNAL, and semaphores (P, V, and mutex)
- Synchronize processes using hardware and software mechanisms

Summary (cont'd.)

88

- Avoid typical problems of synchronization
  - Missed waiting customers
  - Synchronization of producers and consumers
  - Mutual exclusion of readers and writers
- Concurrent processing innovations
  - Threads and multi-core processors
    - Requires modifications to operating systems