M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

Course Name: Operating Systems: CS51
Credits: 3:1:0

Term: September – December 2020

Faculty:
Chandkrika Prasad
Vandana S Sardar

Process Concept

The contents in this presentation are selected from
Operating Systems Concepts – 9th Edition, Silberschatz, Galvin and Gagne @2013

RAMAIAH
Institute of Technology

Operations on Processes

System must provide mechanisms for:

- Process creation
- Process termination
- And so on as detailed next

RAMAIAH
Institute of Technology

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 options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Fork() system call

System call fork() is used to create processes.

The purpose of fork() is to create a new process, which becomes the child process of the caller.

After a new child process is created, both processes will execute the next instruction following the fork() system call.

Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

If fork() returns a negative value, the creation of a child process was unsuccessful.

fork() returns a zero to the newly created child process.

fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h

Fork() system call

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main(int argc, char *argv[]) {
6    printf("hello world (pid:%d)\n", (int) getpid());
7    int rc = fork();
8    if (rc < 0) {
9      // fork failed
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) {
13     // child (new process)
14     printf("hello, I am child (pid:%d)\n", (int) getpid
15   } else {
16     // parent goes down this path (main)
17     printf("hello, I am parent of %d (pid:%d)\n",
18             rc, (int) getpid());
19   }
20   return 0;
21 }
```

Fork() system call

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

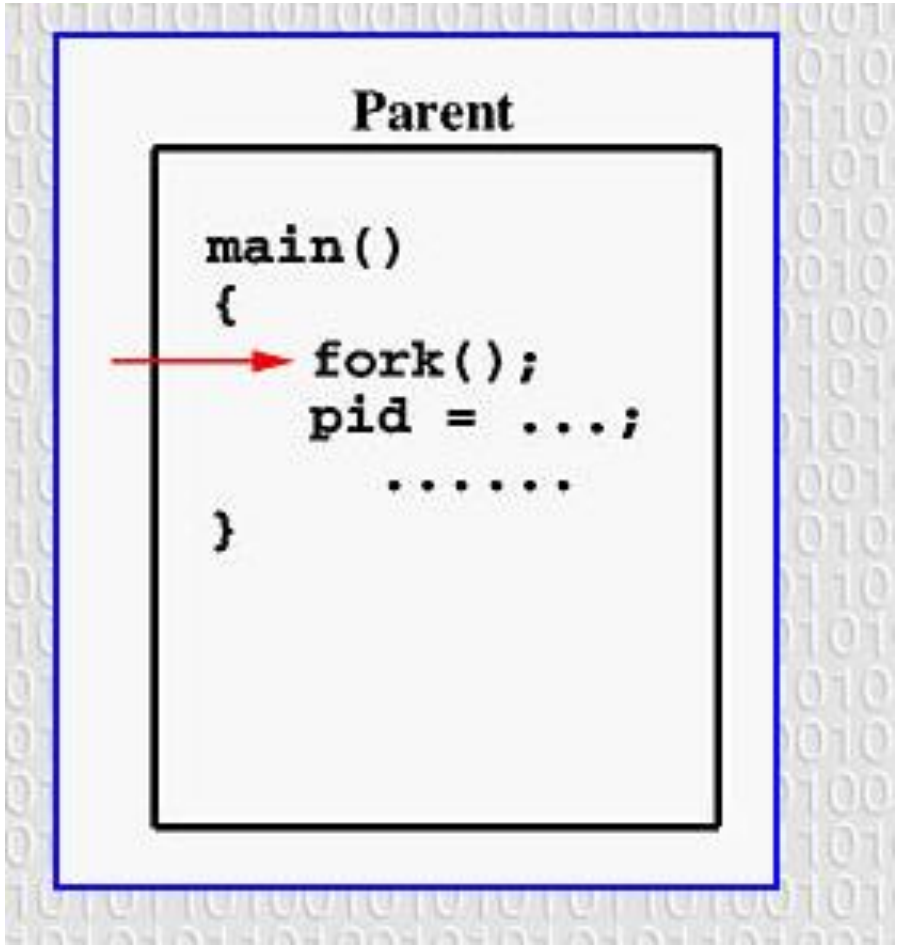
Fork() system call

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(int argc, char *argv[]) {
7    printf("hello world (pid:%d)\n", (int) getpid());
8    int rc = fork();
9    if (rc < 0) {          // fork failed; exit
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) { // child (new process)
13     printf("hello, I am child (pid:%d)\n", (int) getpid());
14   } else {               // parent goes down this path (main)
15     int rc_wait = wait(NULL);
16     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
17            rc, rc_wait, (int) getpid());
18   }
19   return 0;
20 }
```

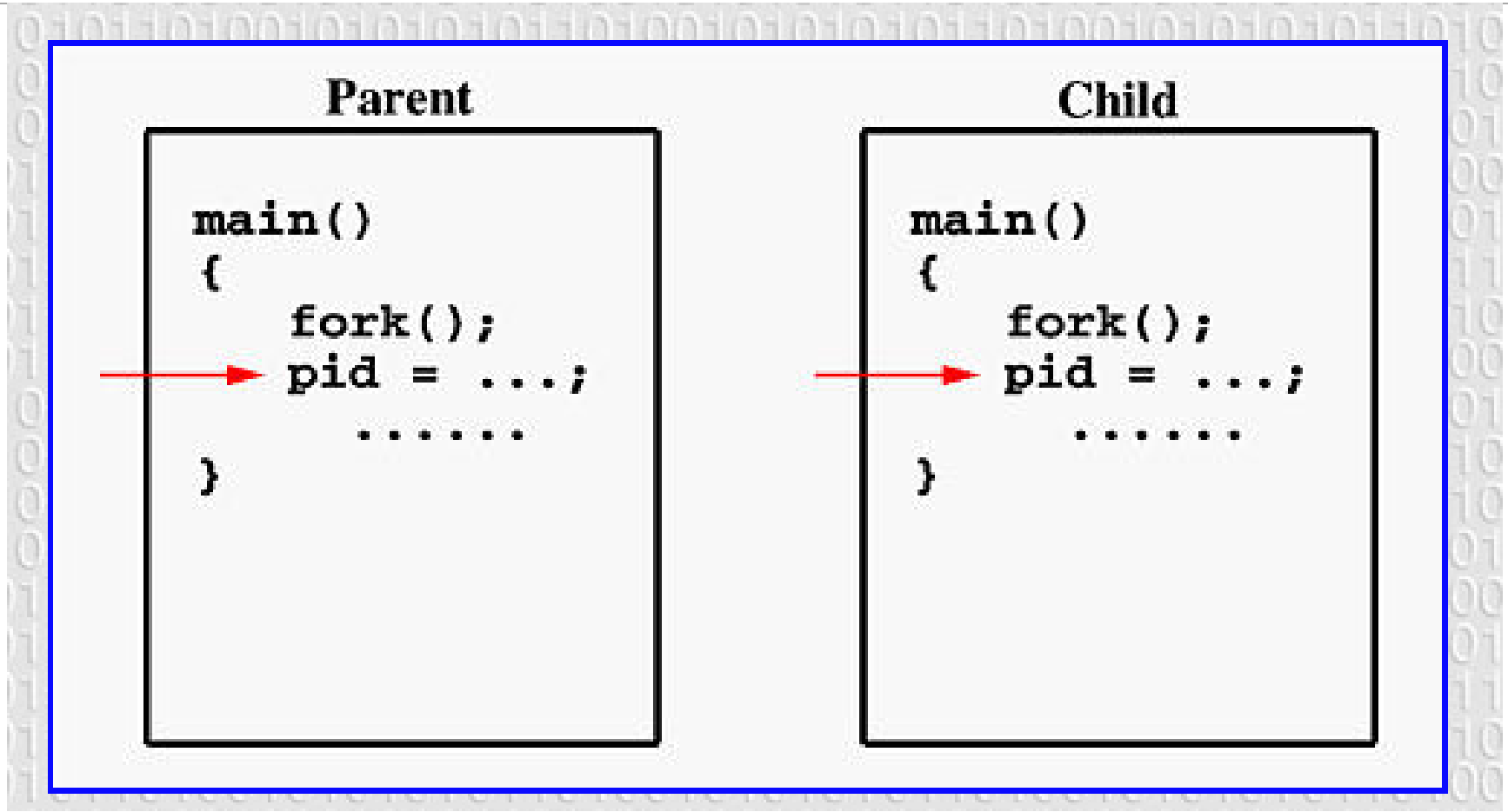
Fork() system call

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

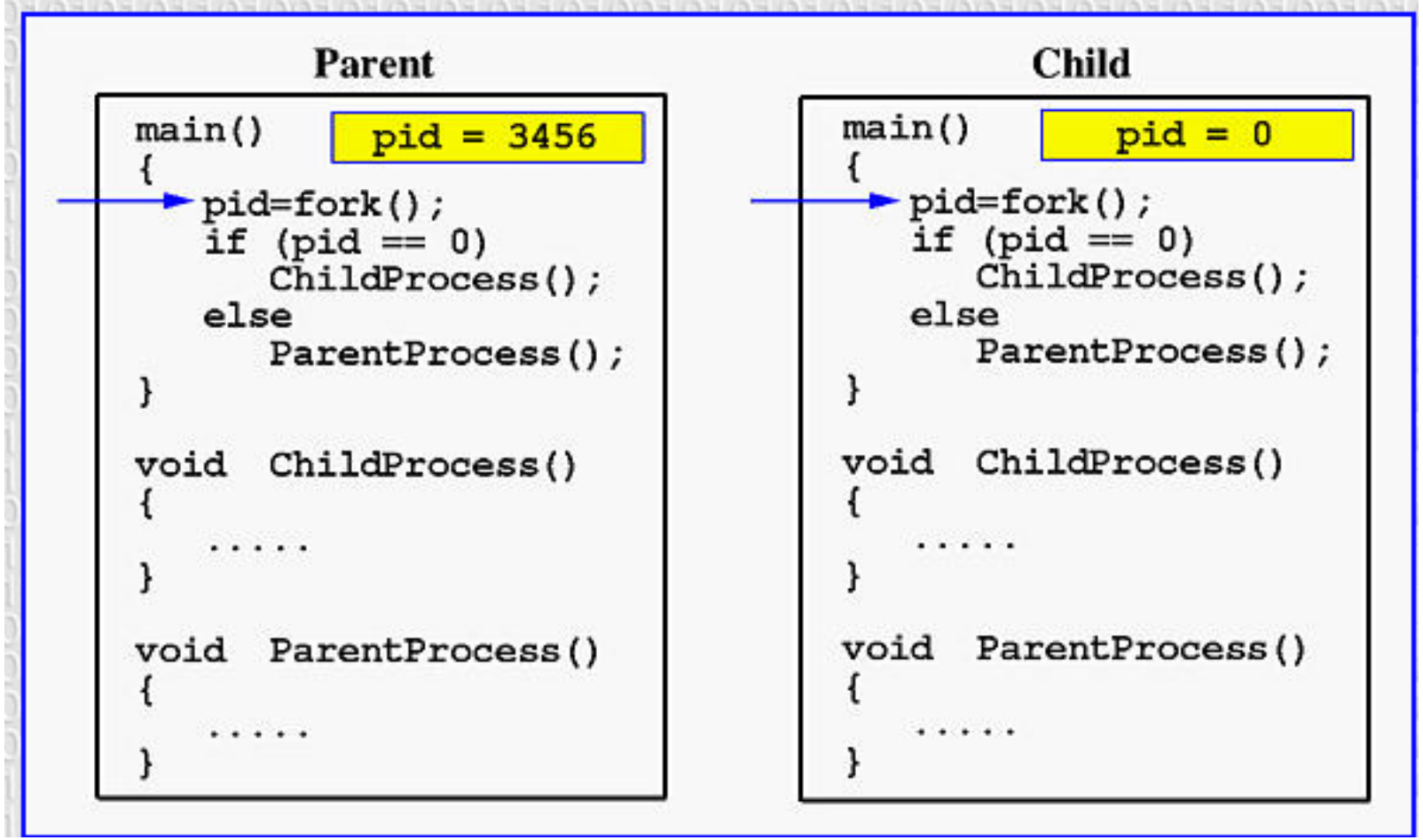
Fork() system call



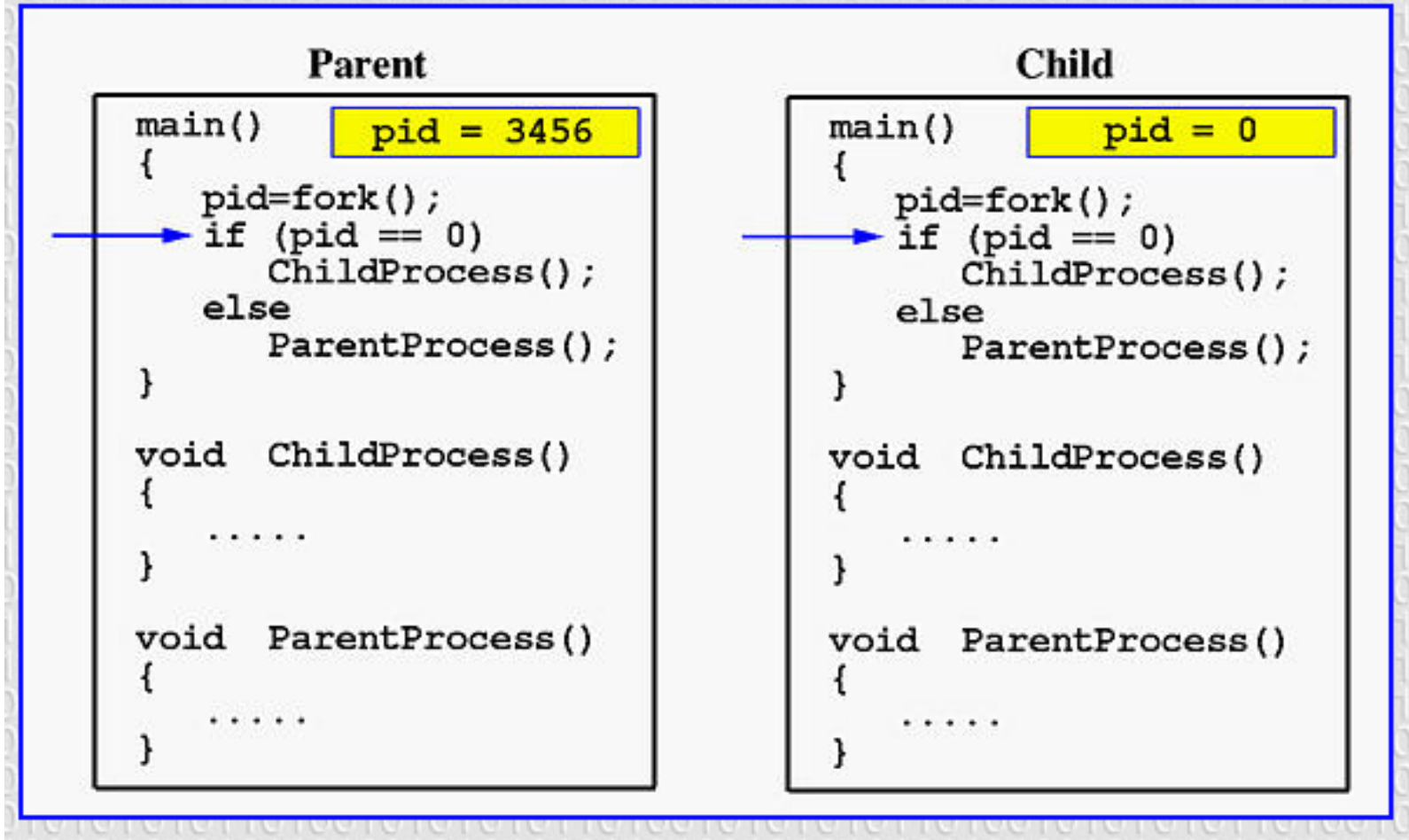
Fork() system call



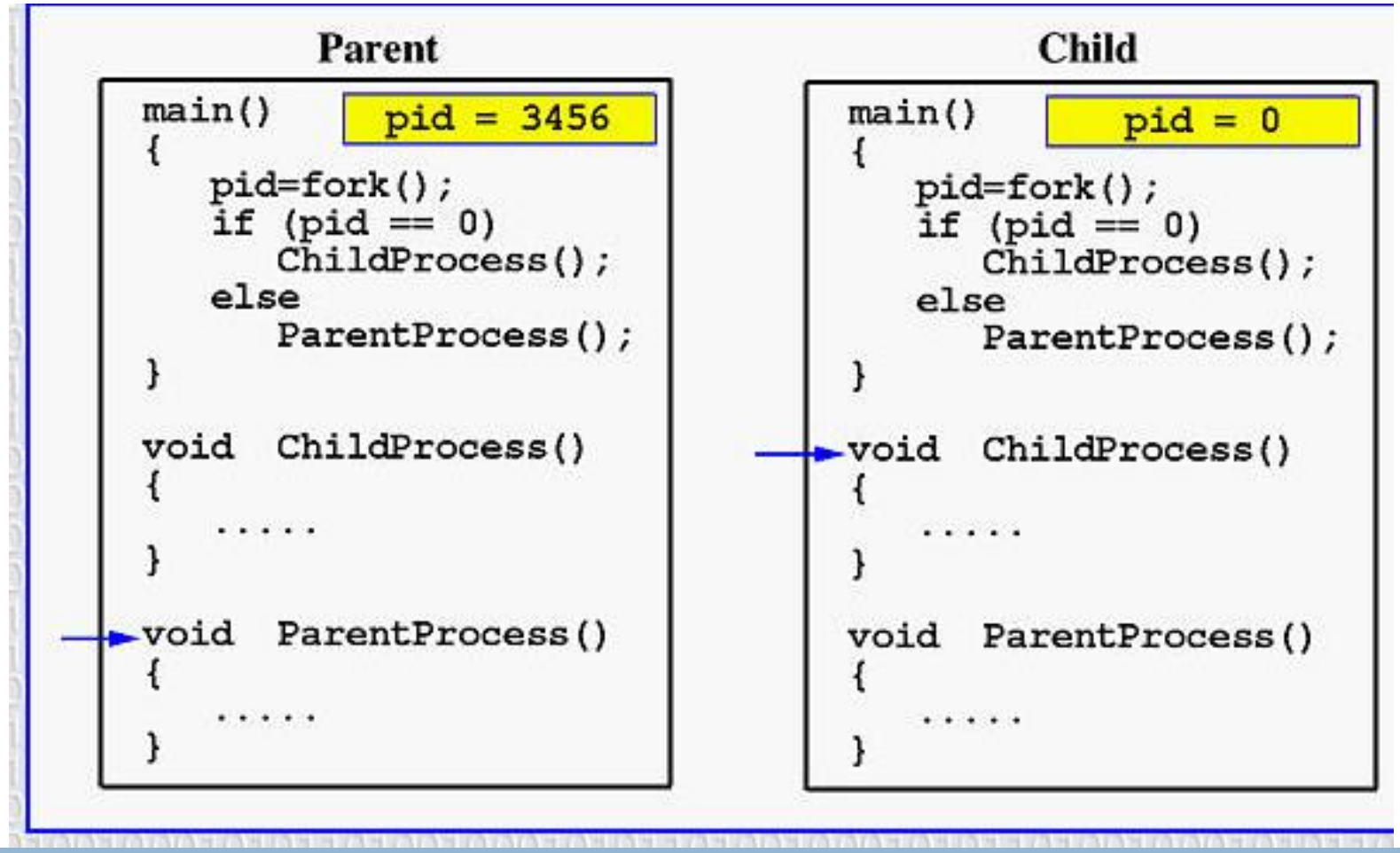
Fork() system call



Fork() system call



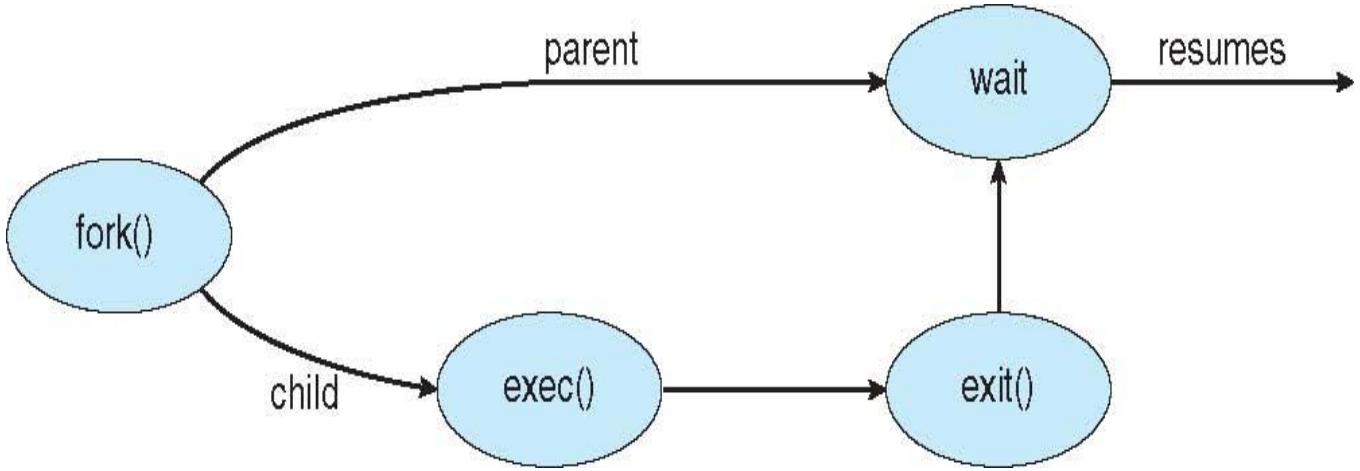
Fork() system call



RAMAIAH
Institute of Technology

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



RAMAIAH
Institute of Technology

C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Output??

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

```
Hello world!
Hello world!
```

Output??

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

```
hello
hello
hello
hello
hello
hello
hello
hello
```

Explanation of output

```
fork ();   // Line 1
fork ();   // Line 2
fork ();   // Line 3

       L1       // There will be 1 child process
    /     \     // created by line 1.
  L2      L2    // There will be 2 child processes
 /  \    /  \   //  created by line 2
L3  L3  L3  L3  // There will be 4 child processes
                // created by line 3
```

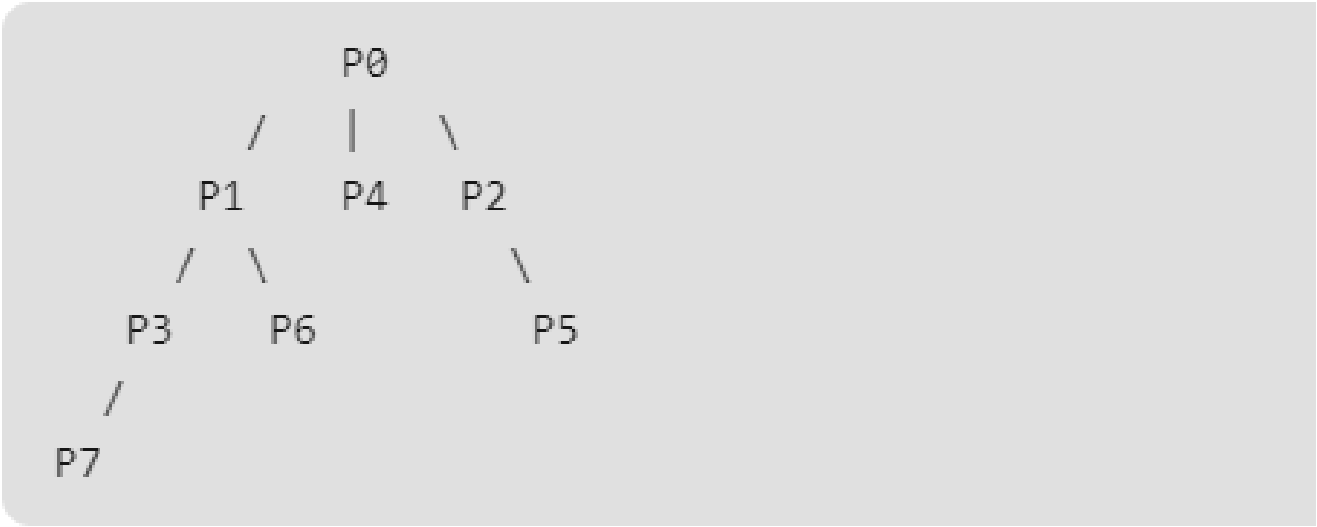
Explanation of output

The main process: P0

Processes created by the 1st fork: P1

Processes created by the 2nd fork: P2, P3

Processes created by the 3rd fork: P4, P5, P6, P7



Output??

```
include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
if (fork() == 0)          // child process because return value zero
       printf("Hello from Child!\n");
     // parent process because return value non-zero.
   else
       printf("Hello from Parent!\n");
}

int main()

{    forkexample();

   return 0;

}
```

```
1.
Hello from Child!
Hello from Parent!
     (or)
2.
Hello from Parent!
Hello from Child!
```

Output??

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}

int main()
{
    forkexample();
    return 0;
}
```

```
Parent has x = 0
Child has x = 2
     (or)
Child has x = 2
Parent has x = 0
```

RAMAIAH Institute of Technology

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process's resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

RAMAIAH
Institute of Technology

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

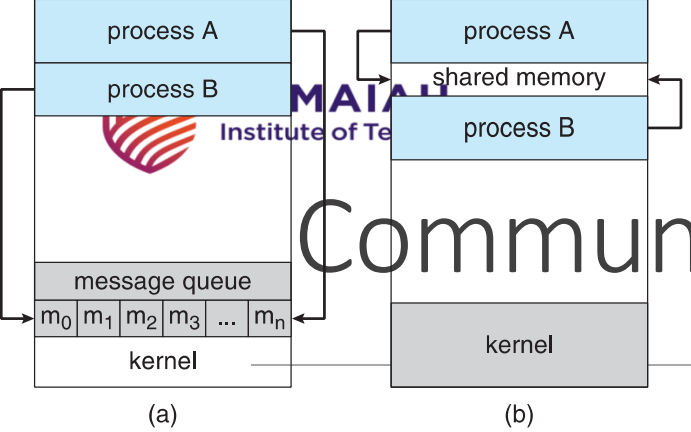
- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait`**, process is an **orphan**

RAMAIAH
Institute of Technology

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models



(a) Message passing. (b) shared memory.

RAMAIAH
Institute of Technology

Cooperating Processes

- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Producer-Consumer Problem

Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process

- **unbounded-buffer** places no practical limit on the size of the buffer
- **bounded-buffer** assumes that there is a fixed buffer size

RAMAIAH
Institute of Technology

Bounded-Buffer – Shared-Memory Solution

Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Solution is correct, but can only use BUFFER_SIZE-1 elements

RAMAIAH
Institute of Technology

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

RAMAIAH Institute of Technology

Bounded Buffer – Consumer

```
item next_consumed;

while (true) {
        while (in == out)

                ; /* do nothing */
        next_consumed = buffer[out];

        out = (out + 1) % BUFFER_SIZE;


        /* consume the item in next consumed */

}
```

RAMAIAH
Institute of Technology

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

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 one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

RAMAIAH
Institute of Technology

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
- 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

RAMAIAH
Institute of Technology

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
- **`send`**(*A, message*) – send a message to mailbox A
- **`receive`**(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- 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.

RAMAIAH Institute of Technology

Synchronization

Message passing may be either blocking or non-blocking

Blocking is considered **synchronous**

- **Blocking send** -- the sender is blocked until the message is received
- **Blocking receive** -- the receiver is blocked until a message is available

Non-blocking is considered **asynchronous**

- **Non-blocking send** -- the sender sends the message and continue
- **Non-blocking receive** -- the receiver receives:
  - A valid message, or
  - Null message

- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

RAMAIAH
Institute of Technology

Synchronization (Cont.)

- Producer-consumer becomes trivial

```
message next_produced;

while (true) {
    /* produce an item in next produced */

send(next_produced);

}
```

```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next consumed */
}
```

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Process Synchronization

The contents in this presentation are selected from
Operating Systems Concepts – 9th Edition, Silberschatz, Galvin and Gagne @2013

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:

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 **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** 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.

RAMAIAH
Institute of Technology

Producer

```
while (true) {
        /* produce an item in next produced */


        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

RAMAIAH
Institute of Technology

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```

RAMAIAH Institute of Technology

Race Condition

`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 "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

RAMAIAH Institute of Technology

Critical Section Problem

Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$

Each process has **critical section** segment of code

- Process may be changing common variables, updating table, writing file, etc
- When one process in critical section, no other may be in its critical section

Critical section problem is to design protocol to solve this

Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

RAMAIAH
Institute of Technology

Critical Section

General structure of process P_i

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

RAMAIAH
Institute of Technology

Algorithm for Process P_i

```
do {

        while (turn == j);

              critical section

        turn = j;

              remainder section

} while (true);
```

RAMAIAH
Institute of Technology

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i 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** - A bound must exist 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
   - No assumption concerning **relative speed** of the ***n*** processes

RAMAIAH Institute of Technology

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

RAMAIAH
Institute of Technology

Peterson's Solution

Good algorithmic description of solving the problem

Two process solution

Assume that the **`load`** and **`store`** machine-language 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

The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process P_i is ready!

RAMAIAH
Institute of Technology

Algorithm for Process P_i

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j);
                critical section
        flag[i] = false;
                remainder section
} while (true);
```

RAMAIAH
Institute of Technology

Peterson's Solution (Cont.)

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

   P_i enters CS only if:

   either **`flag[j] = false`** or **`turn = i`**

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

RAMAIAH Institute of Technology

Synchronization Hardware

Many systems provide hardware support for implementing the critical section code.

All solutions below based on idea of **locking**

- Protecting critical regions via locks

Uniprocessors – could disable interrupts

- Currently running code would execute without preemption
- Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable

Modern machines provide special atomic hardware instructions

- **Atomic** = non-interruptible
- Either test memory word and set value
- Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
        acquire lock

                critical section

        release lock

                remainder section

} while (TRUE);
```

RAMAIAH
Institute of Technology

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
  {
        boolean rv = *target;
        *target = TRUE;
        return rv:
  }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

RAMAIAH
Institute of Technology

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
            /* critical section */
    lock = false;
            /* remainder section */
} while (true);
```

RAMAIAH
Institute of Technology

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter "value"
3. Set the variable "value" the value of the passed parameter "new_value" but only if "value" =="expected". That is, the swap takes place only under this condition.

RAMAIAH
Institute of Technology

Solution using compare_and_swap

Shared integer "lock" initialized to 0;

Solution:

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

   /* critical section */

lock = 0;

   /* remainder section */

} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)

      key = test_and_set(&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);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
   acquire() {
      while (!available)

         ; /* busy wait */

      available = false;

   }

   release() {

      available = true;

   }

   do {
   acquire lock

      critical section

   release lock

     remainder section

} while (true);
```

Semaphore

Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.

Semaphore ***S*** – integer variable

Can only be accessed via two indivisible (atomic) operations

- **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**

Semaphore

Definition of the **wait() operation**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

Definition of the **signal() operation**

```
signal(S) {
    S++;
}
```

Semaphore Usage

Counting semaphore – integer value can range over an unrestricted domain

Binary semaphore – integer value can range only between 0 and 1

- Same as a **mutex lock**

Can solve various synchronization problems

Consider P_1 and P_2 that require S_1 to happen before S_2

Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

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

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

- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied

Note that applications may spend lots of time in critical sections and therefore this is not a good solution

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

Two operations:

- **block** – place the process invoking the operation on the appropriate waiting queue
- **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

```
typedef struct{
  int value;
  struct process *list;
  } semaphore;
```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {

   S->value--;

   if (S->value < 0) {
      add this process to S->list;

      block();

   }

}

signal(semaphore *S) {

   S->value++;

   if (S->value <= 0) {
      remove a process P from S->list;

      wakeup(P);

   }

}
```

RAMAIAH Institute of Technology

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

Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Starvation – **indefinite blocking**

- A process may never be removed from the semaphore queue in which it is suspended

Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

- Solved via **priority-inheritance protocol**

RAMAIAH
Institute of Technology

Classical Problems of Synchronization

Classical problems used to test newly-proposed synchronization schemes

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

RAMAIAH
Institute of Technology

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

RAMAIAH Institute of Technology

Bounded Buffer Problem (Cont.)

The structure of the producer process

```
do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
} while (true);
```

RAMAIAH Institute of Technology

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
} while (true);
```

RAMAIAH Institute of Technology

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 write

Problem – allow multiple readers to read at the same time

- Only one single writer can access the shared data at the same time

Several variations of how readers and writers are considered – all involve some form of priorities

Shared Data

- Data set
- Semaphore **`rw_mutex`** initialized to 1
- Semaphore **`mutex`** initialized to 1
- Integer **`read_count`** initialized to 0

RAMAIAH
Institute of Technology

Readers-Writers Problem (Cont.)

The structure of a writer process

```
do {
    wait(rw_mutex);

        ...
    /* writing is performed */

        ...

   signal(rw_mutex);

} while (true);
```

RAMAIAH Institute of Technology

Readers-Writers Problem (Cont.)

The structure of a reader process

```
do {
     wait(mutex);
     read_count++;
     if (read_count == 1)

       wait(rw_mutex);

    signal(mutex);

       ...
     /* reading is performed */

       ...

    wait(mutex);
     read_count--;
     if (read_count == 0)

    signal(rw_mutex);

    signal(mutex);

} while (true);
```

Readers-Writers Problem Variations

First variation – no reader kept waiting unless writer has permission to use shared object

Second variation – once writer is ready, it performs the write ASAP

Both may have starvation leading to even more variations

Problem is solved on some systems by kernel providing reader-writer locks

RAMAIAH
Institute of Technology

Dining-Philosophers Problem

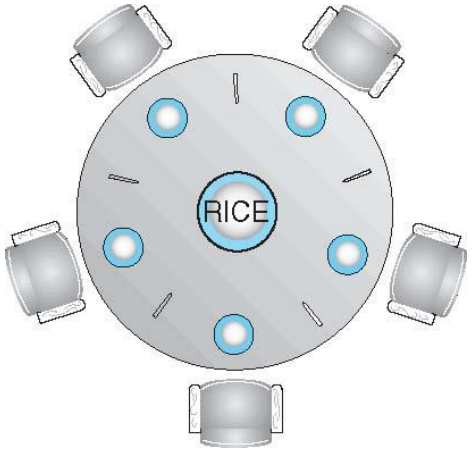
Philosophers spend their lives alternating thinking and eating

Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl

- Need both to eat, then release both when done

In the case of 5 philosophers

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1



RAMAIAH
Institute of Technology

Dining-Philosophers Problem Algorithm

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);
```

What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

Deadlock handling

- Allow at most 4 philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
- Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

RAMAIAH
Institute of Technology

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)

Deadlock and starvation are possible.