

## CHAPTER 2

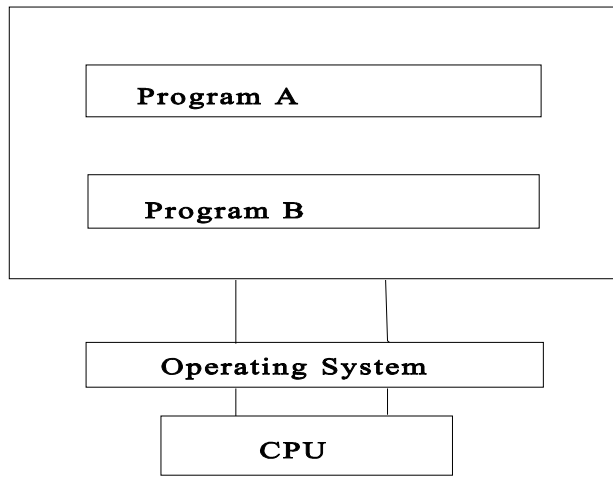
### SHARED MEMORY PROCESSING

#### 1.1 Multiprocessing

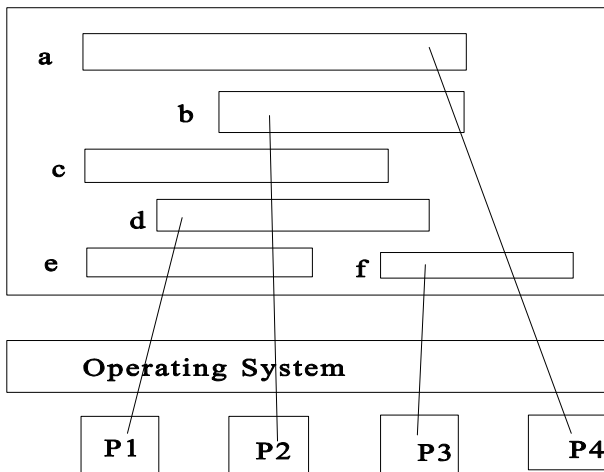
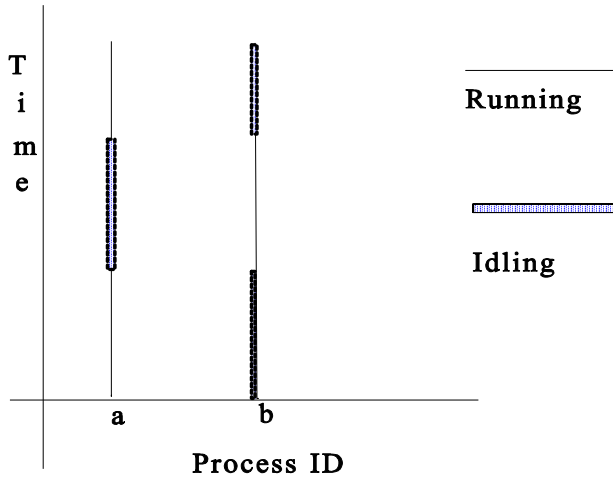
For the remainder of these notes, we assume that a multiprocessing, shared memory, synchronous computer with several processors, similar to *MARS* (*parallel computer with six processors*), is available. Such a machine is often also known as multiuser, multitasking computer. In such an environment, the operating system (OS) can serve several users (hence multiuser) who may have one or more programs running (hence multitasking). UNIX is an example of such an operating system. The OS controls the regions of memory the various CPUs can access. The processors communicate through common memory areas. Parallel programming on such a machine would then require careful attention to load balancing, contention etc., discussed under **MIMD** architectures.

To understand parallel processing better, consider the case of a computer with a single CPU running on a multiuser, multitasking OS. It will soon become clear (when we discuss *processes*) that from the point of view of programming, it is immaterial whether we have a single or multiprocessor computer. Parallel programs can be written on a single CPU machine; however, because of overhead involved, they would run quite slow with no advantage gained. In such a computer, at a given moment, one or more users have programs running, all of which have not terminated. The function of OS is to ensure that all programs have access to the CPU and that the CPU accesses only the memory of the executing program and not any other.

**Example:** The diagram on the right illustrates a case with two programs A and B "running". At a given instant, only A or B is running. After A runs for a while, the operating system, on its own "idles" A and switches to B. When B is running, no reference may be made to A. (In UNIX, a program can be in several modes: asleep, ready-to-run, swapped-to-disk, etc.).

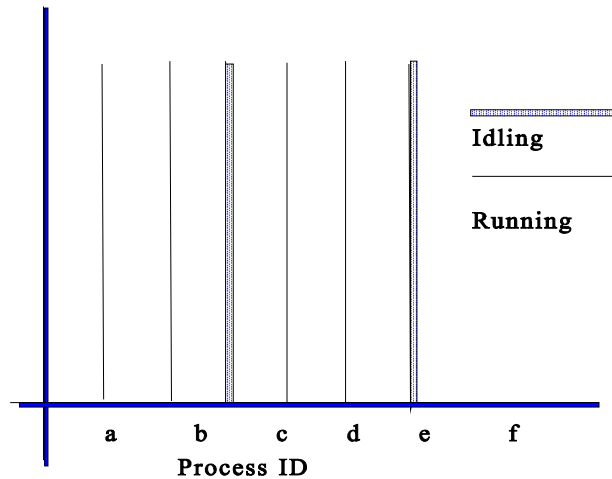


Time line diagrams can be drawn to indicate running status of programs. In our diagram, solid lines indicate running programs while broken (wavy rectangular) lines indicate idling programs. It is important to observe that the operating system has to manage other functions as well, such as I/O.

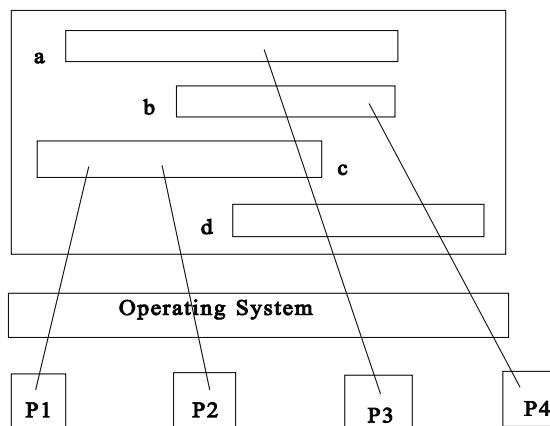


In the case of a computer with several processors, the situation is not very much different (see diagram on left). When there are several programs, each program is run on a separate processor. If there are not enough processors to run all the programs, some would be idling. In this model, no processor is favored over the other. The operating

system schedules the programs to run on any available processor, and may itself be running on a processor. In the example, programs c and e are idle. The other four programs a, b, d, and f are using the various four processors as shown (e.g., a is running on P4). A corresponding time-line diagram is shown on the right. It is important to note that a CPU can run *only one program at a time*.

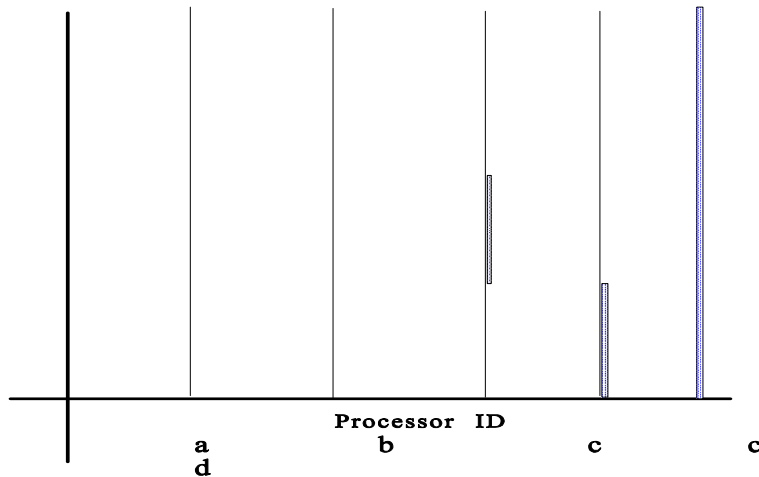


### Parallel programs



In this case, a single program which contains code that can be apportioned among several processors is run on one or more processors. The OS may be running other programs as well (including itself). When a single program is run on more than one processor, each processor does only part of the calculation. In the following example, a machine

with four processors is running four programs. One program, c, is actually run on two processors, P1 and P2. Of the others, a is run on P3, b on P4, and d is idling. Notice again that a CPU can run only one program at a time.



The time-line diagram is shown on the right. It is important to note that while c is being run on two processors P1 and P2, some data may be shared, some private. When sharing data, they can both access the same memory location but only one at a time.

## Program splitting

How do we split a program into two or more parts? Consider the following example in pseudocode:

```
float Sum, a(1000), b(1000);
int n, i ;
{ some program code)
Sum = 0;
for (i= 0; i<n;i++)
    Sum = Sum + a(i)*b(i);
{ more program code }
```

We can split this (serial) program among two processors as follows:

***Processor 1***

```
Sum1 = 0;
for (i=0; i<n;i=i+2)
    Sum1 = Sum1 + a(i)*b(i);
```

.....

***Processor 2***

```
Sum2 = 0;
for (i= 1;i<n;i=i+2)
    Sum2= Sum2 + a(i)*b(i);
```

.....

When done, one of the processors can add the two partial sums:

```
Sum = Sum1 + Sum2;
```

Note that we could get a speedup of close to 2.

In summary, (i) each CPU of a multiprocessor can be modeled by von Neumann serial model, (ii) each CPU can access any memory location but the same location cannot be accessed by two CPUs simultaneously, (iii) the operating system switches and schedules among available processor(s), and that a CPU can access only one program at a time, and (iv) program scheduling is random.

## **1.2 Processors, Processes, Shared Memory**

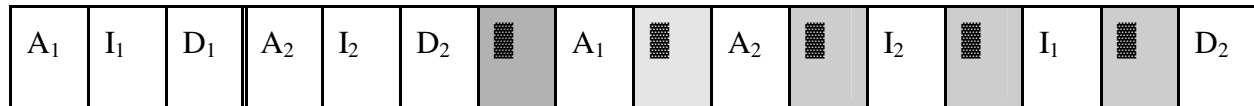
One way to create parallel programs is by apportioning a single calculation among several processors so that, to the operating system, each appears to be an independent program. In such an environment, they may share some data. This is done by means of *processes*.

A processor is a hardware unit. A multiprocessing computer may have one or more processors. On the other hand, a process is a generalization of the notion of a program. It contains (in UNIX):

- a. the program instructions to complete the job (I),
- b. space set aside for datum needed to complete the instructions (D),

c. administrative details such as, page tables, open files, pointers to stacks etc. (A).

When a user executes a program, the operating system creates a corresponding process in memory.

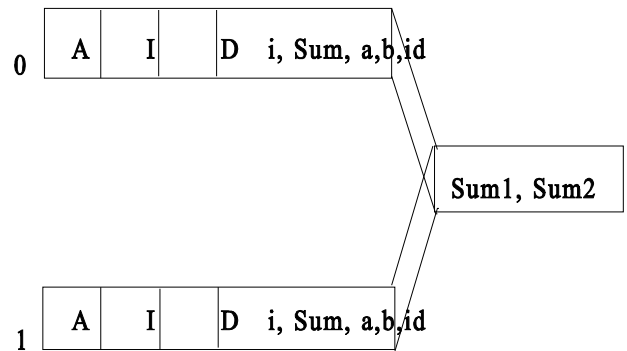


*Logical view of processes*

*Physical view of processes*

We can now define a process as a program with its supporting A, I, D structures. The *Administrative section* of a process has complete information the operating system needs to execute the process, such pointers to instruction and data sections, information on open files, stack, page tables, and other information to time share with other processes (including itself). The *Instruction section* contains complete machine code to run or execute the process. The *Data section* contains values of global variables. With this definition, it is now clear that when the operating system "runs" a program, it is merely executing the associated process. Several processes can be run on a serial machine as well, except that we cannot run any more than one at a time, unless the operating system supports multitasking. In a parallel computer with several processors, the operating system simply distributes the various processes amongst the available processors. From a programming point of view, there is really no difference between a uniprocessor and a multiprocessor machine, since both simply execute associated processes. However, there is almost always a difference in performance.

The three sections A, I, D for a process, although logically appearing to be contiguous, need not be so in memory. It is quite likely that they may be fragmented. Notice that the operating system also needs to maintain information to keep track of the status of each process. References to various fragments of the A, I, D sections are mapped by the page tables.



Parallel processes in memory

**Example:** Consider the example of splitting a sum amongst two processes, in page 2-4. One way to visualize how these two processes occupy memory is shown on the right.

Several points are worth noting.

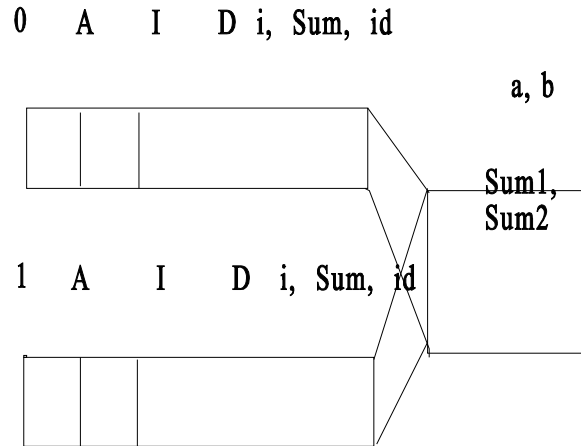
1. The variables `Sum1` and `Sum2` are *shared*. Hence, as soon as `Sum2` is assigned by Process 1, Process 0 can assign `Sum` a value :

$$\text{Sum} = \text{Sum1} + \text{Sum2};$$

2. The identity `id` of each process is private. It helps us determine which portion of the computation is assigned to it.
3. The operating system treats both processes completely separate. Again, it is worth remembering that only one process can access a shared memory at a given time.

4. Each process has its own copy of the arrays a and b.

To conserve resources, we could have them shared as well. In this case, the memory map looks as shown on the right.



### 1.3 Creating Processes - Fork

Processes are created by the operating system. In UNIX, the function call to create a new process from a given process (program) is by means of the function call *fork* :

```
id = fork ( );
```

This creates an additional process, so that including the original process there are two now. The calling process is called the parent (or mother) and the newly created process is called a child (or slave). After a call to *fork*, the parent process is given the *id* 0 while the offspring is given the *id* 1. Note that these are private. It is possible to create several additional processes. In this case, the corresponding call is

```
id = fork (procs)
```

where *procs* is an integer. This creates an additional (*procs* - 1) processes, so that including the calling process there are now *procs* number of processes. In this case, the parent gets the *id* 0, while the children are numbered 1, 2,..., *procs*-1. Note the difference in *id* numbering when we have more than 2 processes. If *procs* =1, there is only one process, namely the parent, so that no new processes would be created.

Because Fork is a UNIX system call, its code is in C. The source code is given in the Appendix. Note that the function Fork returns integers, namely the id-s of the various processors, including the parent.

### Example using Fork:

```
//To build, type:      g++ forkagain.cpp -o again util.o
//util.o must be in the working directory
#include <iostream.h>

extern "C"{                //header files for c fuctions
in util.o
int process_fork(int nproc);
void process_join(int nproc,int id);
}

main()
{
    int i, id, procs;

    cout << "Here we go again \n";
    procs = 3;
    id = process_fork(procs); //create procs - 1 additional processes

    switch(id){
    case 0:
        cout << "Mother - id = " << id << endl;           //each process
prints results
        process_join(procs, id);
        break;
    case 1:
    case 2:
        cout << "Child - id = " << id << endl;
        process_join(procs, id);                          //kill the
children
        break;
    }

    cout << "In the end, I'm still here, id = " << id << endl;

    return 0;
}
```

If this program is run on a uni (or multi-) processor machine, one possible output will be:

```
Hi! I am the big Mama, and you know my ID is 0
Hi! I am the big Mama, and you know my ID is 0
And I am a baby, my ID is 1
And I am a baby, my ID is 1
```

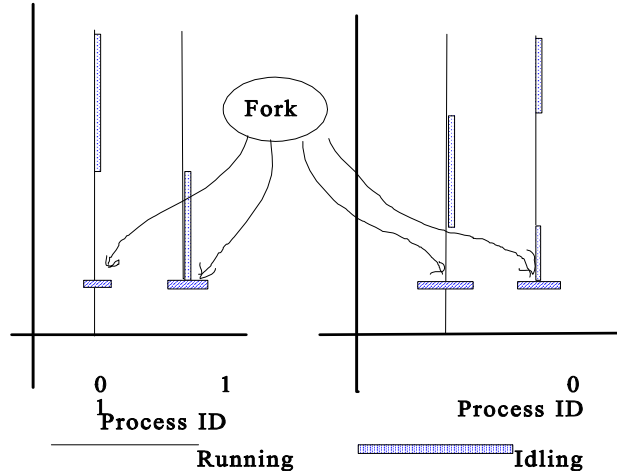
We said this is one possible outcome! In fact, the output can be a permutation of any of the four lines! Remember to include the `<<endl` at the end of each line that is output by a process. This will flush out the output buffer. Otherwise, in

addition to be the interleaved output there will

be a mixup of characters from the different lines of output (a big mess!). The reason for this apparent strange behavior is that the operating system can idle any of the process at will. The output obtained depends on the currently active processes. This phenomenon

is better used by observing the time-line

graphs. The output given above corresponds to the first time-line graph on the right. Notice that immediately after the forking, the child process is idled. (Exercise: What is the output corresponding to the second time-line graph?)



The program also contains another function,

```
void process_join(int nproc,int id);
```

This will be explained shortly. For now, think of it as the opposite of forking. Its purpose is to destroy the processes created by fork.

### Comments on `process_fork`

1. `Process_fork` (`procs`) has an integer argument, with `procs >= 1`. It returns *procs-1* additional processes. The calling process has id 0 and the created processes have id-s 0, 1, 2... `procs-1`. Notice that `fork` is executed only by the calling process but not by the children, hence is not an executable statement. (Reason: If it were, forking

would create many more than procs-processes.)

2. `Process_fork` is a function which returns integer values, namely the id-s of the process created.
3. Each new process is a complete copy of the original process including all variables. However in the case of child processes, any changes to the variables are local and hence temporary, unless they are shared.
4. From a logical point of view, only instructions between the calls to `process_fork` and `process_join` really matter (and the only ones executed). The reason is that they were non-existent before a call to `process_fork`, and are destroyed after a call to `process_join`. Nevertheless, all processes have a copy of all procedures and functions, variables, declared prior to the call to fork. Each process is complete with its A,I,D section and ready to run on a processor.

### **Comments on `process_join`**

The other function in the program is `void process_join(int nproc,int id);`. Its role is roughly opposite to that of function `fork`, in that it is called to eliminate the children created by `fork`.

1. `process_join` is a function with two integer arguments, viz., `nproc` (the total number of processes created) and `id`, the individual id of the process calling it. Notice that it does not return any values.
2. When `process_join` is called by a child process, *it is immediately eliminated* (it is the kiss of death!). When called by the parent process, *further execution halts until all the child-processes have been destroyed*.

3. For practical purposes, one can assume that once a process is destroyed, its memory is immediately freed. The operating system no longer has a record of the destroyed processes.
4. `process_join` is an executable statement and each process has a copy of it in its I section (but not of function `fork`). Notice that the second argument is the id of the calling procedure.
5. Logically, it is convenient to think of a counter memory shared by all processes. This is set to `procs` in the beginning. Each time `process_join` is called by a process, it is decremented by one. When the caller is a child, it is also destroyed. When called by a parent, it waits until the counter memory is zero.
6. There is no "fixed place" to position `process_join` in the program and can be placed wherever appropriate. Consider the following program:

```
//To build, type g++ forkagain.cpp -o again util.o
//util.o must be in the working directory
#include <iostream.h>

extern "C"{                                     //header files for c fuctions in util.o
int process_fork(int nproc);
void process_join(int nproc,int id);
}

main()
{
    int i, id, procs;

    cout << "Here we go again \n";
    procs = 3;
    id = process_fork(procs); //create procs - 1 additional processes

    switch(id){
    case 0:
        cout << "Mother - id = " << id << endl; //each process prints results
        process_join(procs, id);
        break;
    case 1:
    case 2:
        cout << "Child - id = " << id << endl;
        process_join(procs, id); //kill the children
        break;
    }
}
```

```
    cout << "In the end, I'm still here, id = " << id << endl;
    return 0;
}
```

One possible output of this code would be

```
Here we go again ...
Mother - id = 0
Child - id = 2
Child - id = 1
In the end, I am still here, id = 0
```

Note that in *all* outputs, the first and last line will be the same; only the three middle lines may be permuted. The reason is obvious. When the first and last lines are output, there are no child-processes.

## Running the program

1. First of all notice that the original program is in C++. Since this would be run on machine running on UNIX, we will have to use a C++ compiler. The codes for functions `process_fork`, and `process_join` are in C, and hence declared as external to the main program. Before running the main program, you must compile the source codes for the C programs. It is usually very convenient to have a single source file containing the codes of `fork` and `join` (say *util.c*). You should compile this with the "-c option":

```
cc -c util.c util.o
```

The file `util.o` is the object file. You can now compile your c++ program (`myprog.cpp`) as follows:

```
g++ -o rpamula_myprog myprog.cpp util.o
```

It may be a wise idea to name the executable code differently from each other in a multi

user environment. Assuming there are no errors, your executable code would be in the file `rpamula_myprog`. You run the program by typing `rpamula_myprog` at the UNIX prompt.

## 1.4 Granularity

The term *granularity* refers to the amount of execution time of each process in the context of parallel processing. It takes considerable amount of operating system resources to create a process together with its A, I, D sections and dispatch it to an available processor. Before creating several processes then, one should carefully weigh the cost of creating a process against the amount of time spent by it in executing the instructions assigned to it.

Suppose that  $n$  processes are created and that the overhead to create a single process is  $\tau$  units of time. Thus it would take  $n\tau$  to create all the processes. Let  $t$  be the time to execute the code in parallel by all the processors. Clearly, it takes  $n\tau + t$  units of time to create the processes and run them in parallel.

Now, if the entire code is run (serially) by the mother, then the time required is  $nt$ . Creation of parallel processes would be worthwhile only if the serial time is substantially larger than the parallel time, that is  $nt \gg n\tau + t$ . The speedup in this case is  $nt/(n\tau + t)$ . For example, if  $n = 100$ ,  $t = 10$ , and  $\tau = 10$ , then  $n\tau + t$  is 1010, the cost of running the code in parallel. The speedup is 1.01! The reason is that the granularity ( $t$ ) in this case, 10, is small. In some situations, by increasing the granularity, the speedup can be increased.

As we saw in the example on page 2 - 4, loops offer an excellent opportunity to divide execution amongst several processes (this is an often used technique in parallel processing, called *loop-splitting*; it will be considered in detail in a subsequent chapter when we discuss the various techniques for parallel processing). In this case, we can find the optimum number of processors to maximize speedup.

Suppose that we have a *for loop* and assume that:

# loop iterations =  $n$ ,

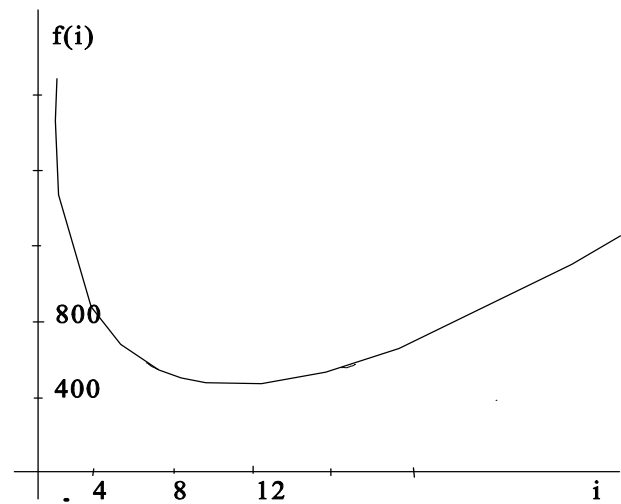
# processes created =  $p$ ,

Time to create one process =  $\tau$ ,

Time to execute a single iteration =  $t$ .

We will assume that  $p * n$ , and that  $n = pi$ .

This means that each processor executes the



code in the loop  $i$  times. Since all the processors work in parallel, the granularity in this case is  $it$ . The cost of creating all the  $p$  processes is clearly  $p\tau$ , so the cost of running the code in parallel is  $f(i) = p\tau + it$ . By substituting  $n/i$  for  $p$  we get  $f(i) = (n\tau/i) + it$ .

To optimize  $i$ , we set  $f'(i) = 0$ , that is  $f'(i) = (-n\tau/i^2) + t = 0$ . Solving this equation, we find the minimum value of  $i = \sqrt{(n\tau/t)}$ . Substituting this in  $f(i)$ , we get

$$f(i) = (n\tau + i^2t)/i = 2n\tau/i = 2t\sqrt{(n\tau)}$$

as the optimal minimum time for execution in parallel. To find the speedup, observe that the serial time for execution is  $nt$ . Hence,

$$\text{optimal speedup} = nt / (2t\sqrt{(n\tau)}) = (\sqrt{2})\sqrt{(n/\tau)}$$

It is interesting that the speedup depends only the total number of loop iterations and the time to create a single process. The granularity in this case *it* since each processor executes the loop body *i* times.

The graph above is drawn assuming  $n = 100$ . The optimal value (in this example) of *i* is 10 when  $\tau = t$ .

## 1.5 Shared variables

Even though `process_fork` creates new processes, there is no communication between the processes. In fact, there cannot be any, since the operating system ensures that the A,I,D sections of the various processes remain hidden from each other. The mechanism that enables processes to share their results is by means of *shared variables*.

Consider the following example:

```
#include <iostream.h>

int procs, id, k;

extern "C"{                                     //header files for external c
functions in util.o
int process_fork(int nproc);
void process_join(int nproc,int id);
}
main()
{
    k =987;
    cout<< "In the beginning.."<<endl;
    cout<< "the value of k is, k = "<<k<<endl;
```

```

    id = process_fork(2);
    if (id==1)
    { k = 1;
      cout << "Child attempted to change k: k = " <<k<<endl;
    }

    process_join (2, id);

    cout << "But apparently did not succeed. k is still " << k << endl;
    return 0;
}

```

When run, the only output of this program will be:

```

In the beginning...
the value of k is, k = 987
Child attempted to change k: k = 1
but apparently didn't succeed. k still is 987

```

The problem is that the variable k is not shared and is private to each process. In the child process it is changed to 1, but lost when the child process is destroyed. Consider another example:

```

#include <iostream.h>

int sum0, sum1, sum;
int procs, id;

extern "C"{
functions in util.o //header files for external c
int process_fork(int nproc);
void process_join(int nproc,int id);
}

main()
{
    procs = 2;
    id = process_fork(procs);
    if (0 == id)
        sum0 = 1+2;
    else
        sum1 = 3+4;
    process_join (procs, id);

    sum = sum0 + sum1;
    cout << "The total of 1 + 2 + 3 + 4 is " << sum << endl;
}

```

```





    return 0;
}

```





The output for this program (unique and no interleaving) would be

The total of 1 + 2 + 3 + 4 is 3



To better understand this phenomenon, consider the memory maps of the two processes, after forking and before process\_join:

P0	A	I	sum=	sum0	sum1	id	P1	A	I	sum	sum0	sum1	id
			0	=0	=0	=0				=0	=0	=0	=1

*Memory map after process\_fork*

P0	A	I	sum=	sum0	sum1	id	P1	A	I	sum	sum0	sum1	id
			0	=3	=0	=0				=0	=0	=7	=1

*Memory map before process\_join*

P0	A	I	sum=	sum0	sum1	id
			3	=3	=0	=0

*Memory map after process\_join*

## Shared memory

Local copies of global variables in child processes are private and temporary. They are not visible to each other and are destroyed when the processes ceases to exist (after executing `process_join`). The mechanism to make variables visible to all the processes is by creating one or more memory addresses that are visible to all the processes.

Shared variables are freely visible to all the processes. Their values can be changed by child processes during execution and all changes are permanent. All changes are visible to the parent process even after the children cease to exist.

In the second example on page 2-17, the final sum would be correct if the variable `sum1` is visible to the mother process, that is, it should be shared by both processes. Notice that this gives the mother access to the child's sum (but not vice-versa) and she can use it in computing the variable `sum`. In UNIX, a variable becomes shared by providing all processes access to a pointer which references the variable. Such a pointer is obtained by issuing an appropriate command to the operating system. The command is actually a function call (in C) and it returns a pointer of the appropriate type. This function must be declared as an external (C) function in your program. Its syntax is:

```
int* sharename (int size, int& shareid ) ;
```

### Remarks

1. *sharename* is a user defined identifier. For example, if the variable to be shared is an integer (as is `sum1` in our example), an appropriate name would be *shareint*. You can share any type of variable, simple or structured.
2. It returns a pointer to the shared memory location.
3. The code for the function is in C. As in the case of `process_fork` and `process_join`,

you must compile it ahead before running your main program. (You may want to have several function calls, one each for real, integer, char, boolean and a general purpose list, all stored in one file.)

4. The function *sharename* has two integer arguments. "Size" is the amount of memory, in bytes, to be allocated (in 32-bit systems, an integer occupies 4 bytes, more on this later). The argument "shareid" is a reference argument. It is assigned an integer value by the Operating System. It will be needed for "clean-up" later.

We return to the example on page 2-17 again this time using shared variables:

```
//program sharemem.cpp translated for R. Pamula by Jim Ivers
//to compile, type g++ sharemem.cpp -o sharemem util.o
//util.o must be in the working directory
//obviously, don't name the program share
#include <iostream.h>

int *p = NULL;           //create a pointer to a type int variable
int sum0, total;
int procs, id;
int shareid;

extern "C"{              //header files for external c
functions in util.o
int process_fork(int nproc);
void process_join(int nproc,int id);
int *shareint (int size, int& id);
void clean_up_shared(int IdShareMem);
}
void add(int *p, int sum);

main()
{
    total = 0;
    p = shareint(4, shareid);           //p points to shared memory
    add (p, sum0);
    total = sum0 + *p;
    cout << "The total of 1 + 2 + 3 + 4 is " << total << endl;

    clean_up_shared(shareid);

    return 0;
}

void add(int *p, int sum){

    procs = 2;
    id = process_fork(procs);
    if (0 == id)
```

```

        sum0 = 1+2;
    else
        *p = 3+4;
    process_join (procs, id);
}

```

## Comments

1. Only one variable of type integer is shared (through the pointer p). Note that \*p is 7 after process\_join is executed, and is still visible to the mother process.
2. Note that the statement

```
p := shareint (4, shareid);
```

makes p itself point to the shared memory region. Also, the first parameter, 4, is the size of \*p (one integer). We can also use the more efficient statement

```
p := shareint (sizeof (*p), 5);
```

This completely avoids having to compute the memory used by \*p.

We consider another example:

```

#include <iostream.h>

const int size = 4;    //For 4 elements per vector
int list[size];
int shareid[4];

extern "C"{
int process_fork(int nproc);           //link c functions from util.o
void process_join(int nproc,int id);
int *shareint (int size, int& id);
int *sharelist (int size, int& id);
}
// header for function dotpro
void dotpro (int *p, int *q, int *sum1, int &dotsum);

main()
{
    //declare main variables
    int j, id = 0, dotsum = 0, *q = list, *p = list, *sum1 = NULL;
}

```

```

p = sharelist (size * sizeof (*p), shareid[0]); //share variables
q = sharelist (size * sizeof (*q), shareid[1]);
sum1 = sharelist (sizeof (*sum1), shareid[2]);

for (j =1; j <= size; j++){          //input vector data
    cout << " p[" << j << "] = ";
    cin >> p[j];
}
for (j =1; j <= size; j++){          //input vector data
    cout << " q[" << j << "] = ";
    cin >> q[j];
}
dotpro ( p, q, sum1, dotsum); //call function to calculate dotproduct
cout << "The dotproduct is " << dotsum << endl; //output result
return 0;
}
//function to calculate the dot product of p and q
void dotpro (int *p, int *q, int *sum1, int &dotsum){
    //declare dotpro variables
    int i, id, procs, temp;

    procs = 2;
    id = process_fork(procs); //create another process
    temp = 0;

    i = id + 1;
    //calculate the dotproduct in parallel
    do{
        temp += (p[i]) * (q[i]); //there are two temps, one for each
process
        cout << "p[" << i << "] = " << p[i] << " q [" << i << "] = " << q[i]
            << " id =" << id << endl;
        i += procs;          //since there are 2 processes, i for the
                            //parent is 1,3,... (odd numbers)
                            //and i for the child is 2,4,...(even numbers)
    }while (i <= size);

    switch (id){
    case 0:
        dotsum = temp;      //the temp for the parent process
        break;
    case 1:
        *sum1 = temp;      //the temp for the child process
    }
    process_join (procs, id); //destroy child process
    dotsum += (*sum1);      //add the parent and child values
                            //and return in dotsum using a reference &
    return;
}

```

This program works. However, we have not released the shared resources (memory pointers).

Before exiting the program, we need to execute the following:

```

clean_up_shared(shareid[0]);
clean_up_shared(shareid[1]);
clean_up_shared(shareid[2]);

```

## 1.5 Contention

A fundamental maxim about the operating system is that all processes are *randomly* scheduled. That is at any given time, a process can be idled and restarted at a later time. It is not necessary that even one process would be scheduled at a given instant. The operating system does this in a way so as to maximize its processing resources. In particular, the amount of time taken to run a process is independent of the process itself; it is quite possible that amongst two processes requiring almost identical amount of work, because of idling, one process may take longer to finish than the other. This gives rise to the phenomenon known as *contention*.

When several processes share a variable, because of operating system idlings, it is possible that the final value of a shared variable may be *changed incorrectly*. In fact the final value of such a variable will very much depend upon the order in which the various processes altered it. The last alteration is permanent. This is better understood by the following example:

```
int i, id;
i : shared; {pseudocode to indicate i is shared}
{
    i = 0;
    id = process_fork (2);
    i = i + 1;
    cout<<"id = "<< id<<" i = "<<i<<endl;
    process_join (2, id)
}
```

When run, this program can produce several outputs. We consider three of those:

```
id = 0 i = 1      id = 1 i = 1      id = 1 i = 1
id = 1 i = 2      id = 0 i = 2      id = 1 i = 1
```

Output 1

Output 2

Output 3

To explain these outputs, consider the main line in the program:

```
i = i + 1;
```

This high level code is equivalent to, in general, the following three assembly language instructions executed by the CPU:

(a) Load value of  $i$  into a CPU register.

(Symbolically,  $R \leftarrow i$ )

(b) Increment register by 1 ( $R^+$ )

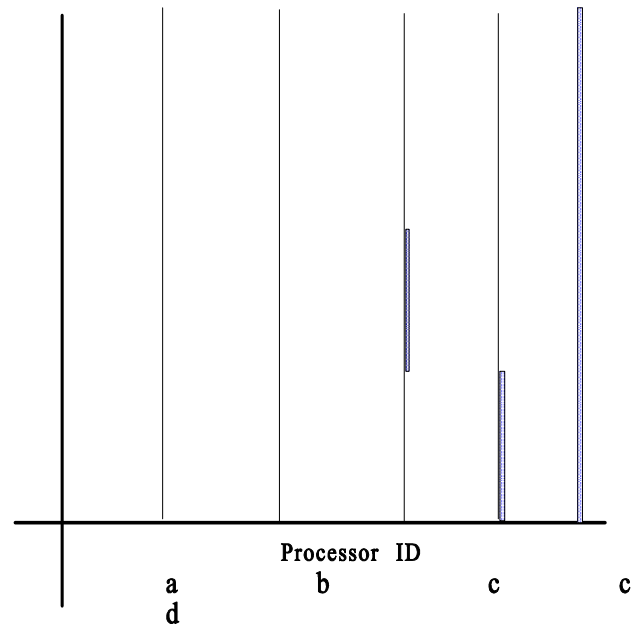
(c) Store  $i$  back in memory ( $i \leftarrow R$ )

Assuming each of these operations is atomic, the operating system can idle a process at the end of any of these steps. In particular, consider the following scenario. This explains the first output.

**FORK K PROCESS 0 PROCESS 1**

Do (a)	$R \leftarrow 0$	idle
Do (b)	$R^+$	idle
Do (c)	$i = 1$	idle
Print values	Do (a) $R \leftarrow 1$	
Wait	Do (b) $R^+$	
Wait	Do (c) $i = 2$	
Wait	Print values	

**JOIN K** Finish program Kill



The time-line diagram on the right explains this better.

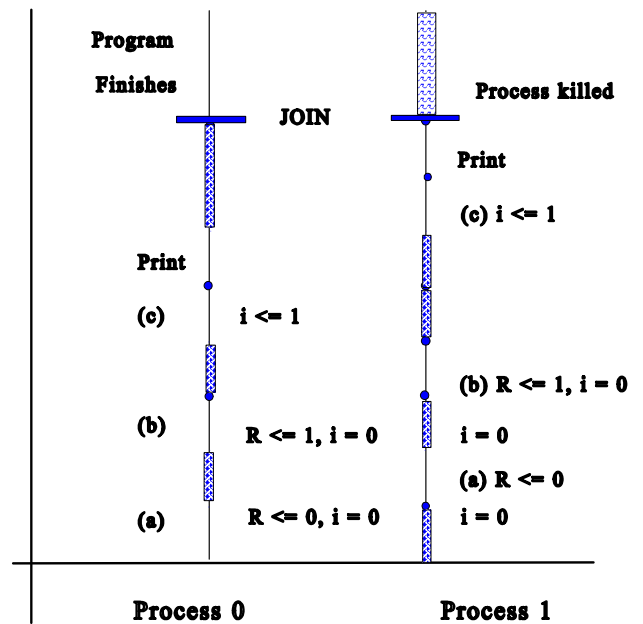
The second output is the result of Processes 0

and 1 reversing their roles.

More interesting is the third output. It is the result of the following scenario:

1. After forking, Process 1 is idled. Process does (a) and then is idled. Value of  $i$  is 0.

2. Process 1 wakes up, does (a), idles. Process 0 wakes up, does (b), and then is idled. Note that the value of *i* is still 0.
3. Process 1 wakes up, does (b), and then is idled. Process 0 wakes up, does (c) and then is idled. Value of *i* has just been changed to 1.
4. Process 1 wakes up, does (c), and then is idled. Value of *i* is still 1. Process 0 wakes up, does the printing and then executes join. It waits for Process 1 to finish.
5. Process 1 wakes up, does the printing, executes join and then dies. Process 0 finishes the program.



Again the time-line diagram illustrates the situation better.

The cause of the problem is the fact that two processes are trying to get access to shared memory and change the value there. One says they are *contending* for the same memory space; hence the name contention.

## 1.6 Memory Clean-up

You will recall that the syntax of the function to create shared memory is

```
int* shname (int size, int& shareid) ;
```

Here *size* denotes the size of the memory required and *shareid* is the shared memory identifier *shmid* in the C code. A pointer is returned which points to the start of the shared memory.

The shared memory identifier *shmid* is allocated by the operating system (UNIX) and maintained in a system table. As you have seen, the role of shared memory is to enable several unrelated processes to communicate to each other. There is a problem with such shared memory

however; *they are not released back to the pool when all processes are finished with them.* The reasoning is that other processes may start up later (from a idle or sleep state) and use them. In short, the operating system does not remember for whom this shared memory was created.

Obviously, there is only a limited number of such shared memory identifiers that are available. Once they have been used up, *no more shared memory can be allocated to other processes.* To avoid this bottleneck, all programs that ask for such memory must issue a command to "clean\_up\_shared". As usual, this is again a UNIX system call in C. You must compile it ahead as in the case of other such procedures. The actual code is in the Appendix. The procedure to be used in your Pascal program is:

```
coid clean_up_shared (int IdShareMem);
```

where IdShareMem is the id of the particular shared memory identifier used. You will call this procedure as the last statement of your program:

```
clean_up (shareid);
```

where shareid is the integer variable used in allocating the shared memory in the first place.

A simpler script/command is available:

```
cleanipc;
```

Since this is a system command, you don't use this as a statement of your program. Instead, *after* you have run your program, at the UNIX prompt, type the above command. It will clean up all shared, inter-process-communication resources. However, we have seen that this script may not work with all resources. You can check for all resources currently used in the system by the command

```
ipcs
```

You will see the user name using the resources. You should release the resources used by you using the system command:

*ipcrm -m xxx*

*ipcrm -s xxx*

where *m* or *s* specifies the type of resource and *xxx* is the id assigned to it.

---

**Important Warning:** Do not leave resources hanging. You must clean up within the program and check again at the system level. When your assignments are executed, I should not see the need to clean up the resources outside of your program. The penalty would be very severe.

---