Homework Assignment #1 Multi-Processes Programming: Matrix Computation

Outline

- **Creating Processes**
- **IPC by Shared Memory**
- **Homework Assignment**

Outline

- **Creating Processes**
- **IPC by Shared Memory**
- **Homework Assignment**

Fork()

- *Fork()*
  - create a child process

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

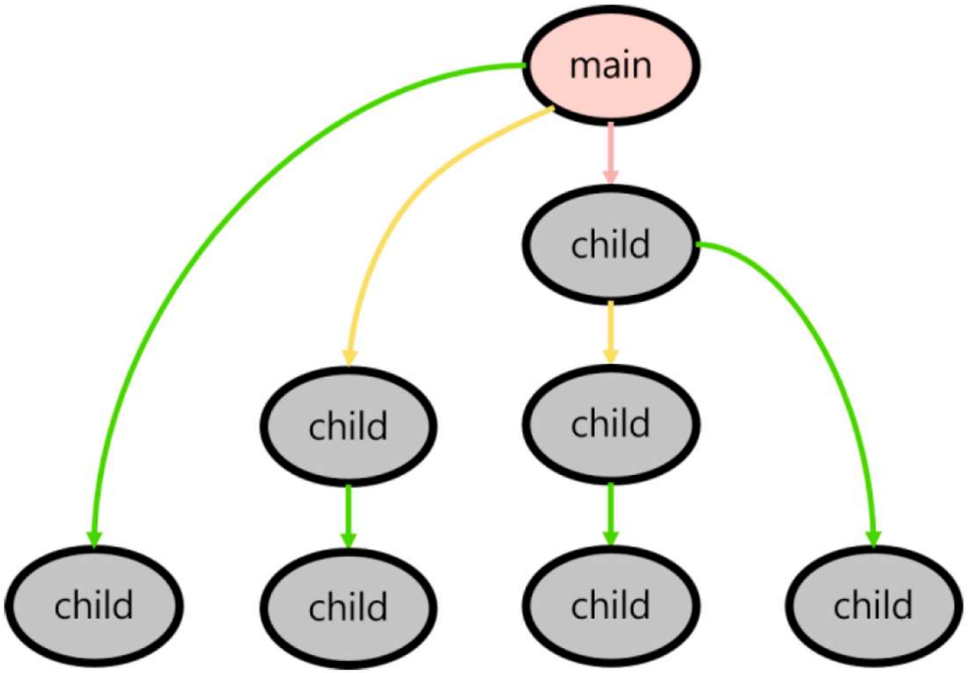
```
int fork();
```

```
EX: int f=fork();
```

- **Describe: *fork()*** creates a new process by duplicating the calling process.
  - The new process is referred to as the *child* process.
  - The calling process is referred to as the *parent* process.
  - The child process and the parent process run in separate memory spaces.
  - At the time of **fork**() both memory spaces have the same content.

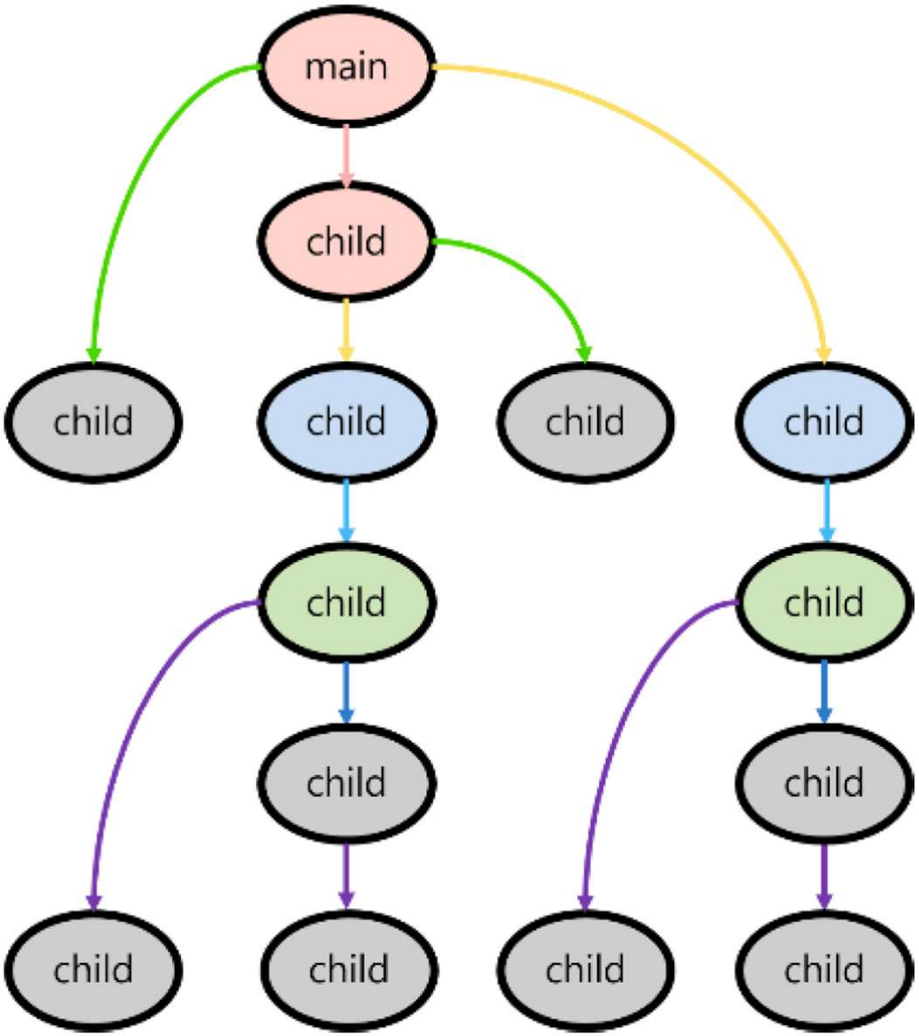
Example

```
int main()
{
        fork();
        fork();
        fork();
}
```

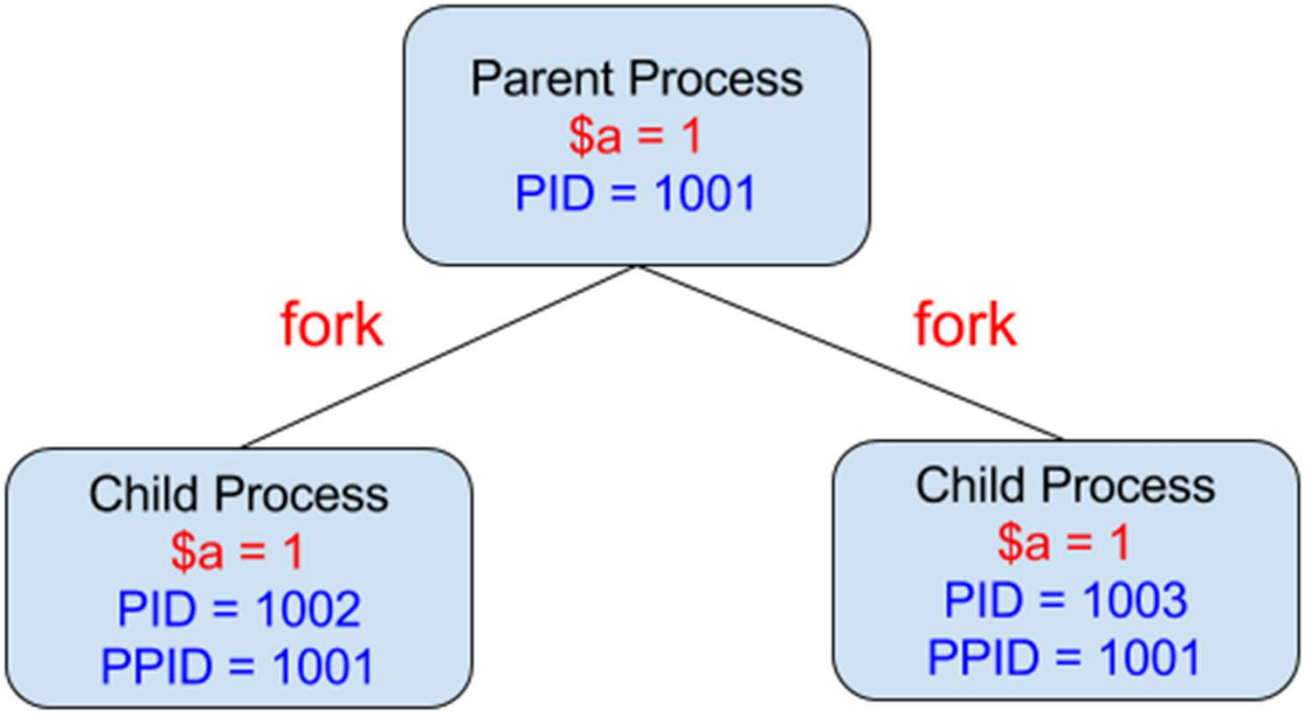


Example(cont.)

```
int main()
{
        fork();
        if(fork()>0)
        {
                fork();
        }
        else if(fork()==0)
        {
                fork();
                fork();
        }
}
```



Example(cont.)



Example(cont.)

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main(int argc,char **argv){
        int i;
        i=fork();
        if(i==0)
                printf("welcome\n");//child process
        else if(i>0)
                printf("you are not welcome!\n");//parent process
        else
                printf(".....\n");//failure


        return 0;
}
~
```

Example(cont.)

```
root@andrew-ubuntu:/home/andrew# gcc -o test test.c
root@andrew-ubuntu:/home/andrew# ./test
you are not welcome!
welcome
```

- Parent & child process may not follow the order of code.
- We don’t know which runs first; it depends on the scheduler.
  - Thus, each execution may have different orders.

wait()

- **wait()**

1. The **wait()** system call suspends execution of the calling thread until one of its children terminates.
2. If *wstatus* is not NULL, **wait()** encoded status information in the *int* to which it points.

```
#include <sys/wait.h>
```

```
pid_t wait(int *wstatus);
```

```
EX:pid=wait(int *k);
```

wait(cont.)

- **WEXITSTATUS**

➢ Decoded and returns the exit status of the child process.

```
#include <sys/wait.h>
```

```
WEXITSTATUS(int wstatus);
```

```
EX:WEXITSTATUS(value);
```

exit()

- exit()
  - The **exit**() function causes normal process termination and the least significant byte of *status* (i.e., *status & 0xFF*) is returned to the parent.

```
#include <sys/wait.h>
```

```
void exit(integer);
```

```
EX: exit(255);
```

Example

```c
#include <stdlib.h>
#include <sys/wait.h>
#include <string.h>
#include <fcntl.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main(){
        int k;
        int f=fork();
        if(f==0){//child process
                exit(255);//sending integer 255
        }
        else{//parent process
                int i,pid;
                pid=wait(&k);//encoded 255,and put in address k
                printf("k=%d(encoded)\n",k);
                i=WEXITSTATUS(k);//decoded information in k
                printf("i=%d(decoded)\n",i);//print 255
        }
```

```
andrew@andrew-ubuntu:~$ ./test
k=65280(encoded)
i=255(decoded)
```

- Notably, the parameter passed in wait() can't be greater than 255.

Outline

- **Creating Processes by *fork()***
- **IPC by Shared Memory**
- **Homework Assignment**

Shared memory API

- ***shm_open()***
  - creates and opens a new, or opens an existing, POSIX shared memory object.

```
#include <sys/mman.h>
#include<fcntl.h>
#include<sys/stat.h>
```

*int shm_open(***const char** ***name*, **int** *oflag*, **mode_t** *mode);*

EX: int fd=shm_open(“OS”, O_CREAT|O_RDWR, 0666);

Shared memory API(cont.)

- ***shm_unlink()***
  - performs the converse operation, removing an object previously created by ***shm_open()***.

```
#include <sys/mman.h>
#include<fcntl.h>
#include<sys/stat.h>
```

*int shm_unlink(***const char** ****name);***

EX: *shm_unlink("OS");*

Shared memory API(cont.)

- ***ftruncate()***
  - truncate a file to a specified length

```
#include <sys/mman.h>
#include<fcntl.h>
#include<sys/stat.h>
```

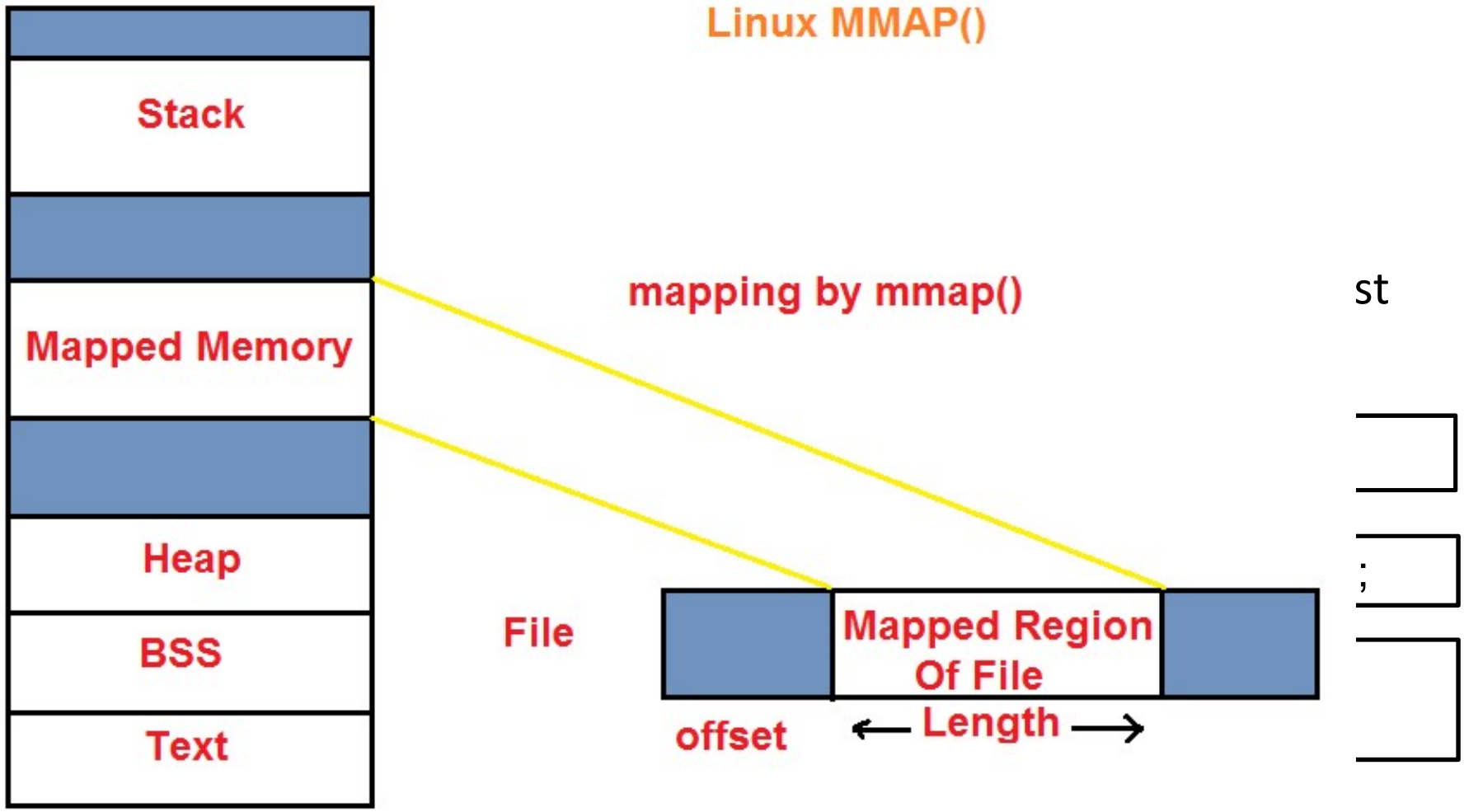
```
int ftruncate(int fildes, off_t length);
```

```
EX: ftruncate(fd,1024);
```

Example

```
const char *name="OS";
int shm_fd;
char buffer[1024];
char *shm_base;
char *ptr;
shm_fd=shm_open(name,O_CREAT|O_RDWR,0666);
if(shm_fd<0){
        perror("open error:\n");
        exit(1);
}
ftruncate(shm_fd,1024);
shm_base=mmap(0,1024,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd,0);
if(shm_base==MAP_FAILED){
        perror("shm_base error:\n");
        exit(1);
}
fgets(buffer,sizeof(buffer),stdin);
ptr=shm_base;
ptr+=sprintf(ptr,"%s",buffer);
if(munmap(shm_base,1024)==-1){
        perror("munmap error:\n");
        exit(1);
}
if(close(shm_fd)==-1){
        perror("close error\n");
        exit(1);
}
shm_unlink("OS");
return 0;
```

Shared memory API(cont.)



Shared memory API(cont.)

- *munmap()*
  - The **munmap**() system call deletes the mappings for the specified address range.
  - The region is also automatically unmapped when the process is terminated.

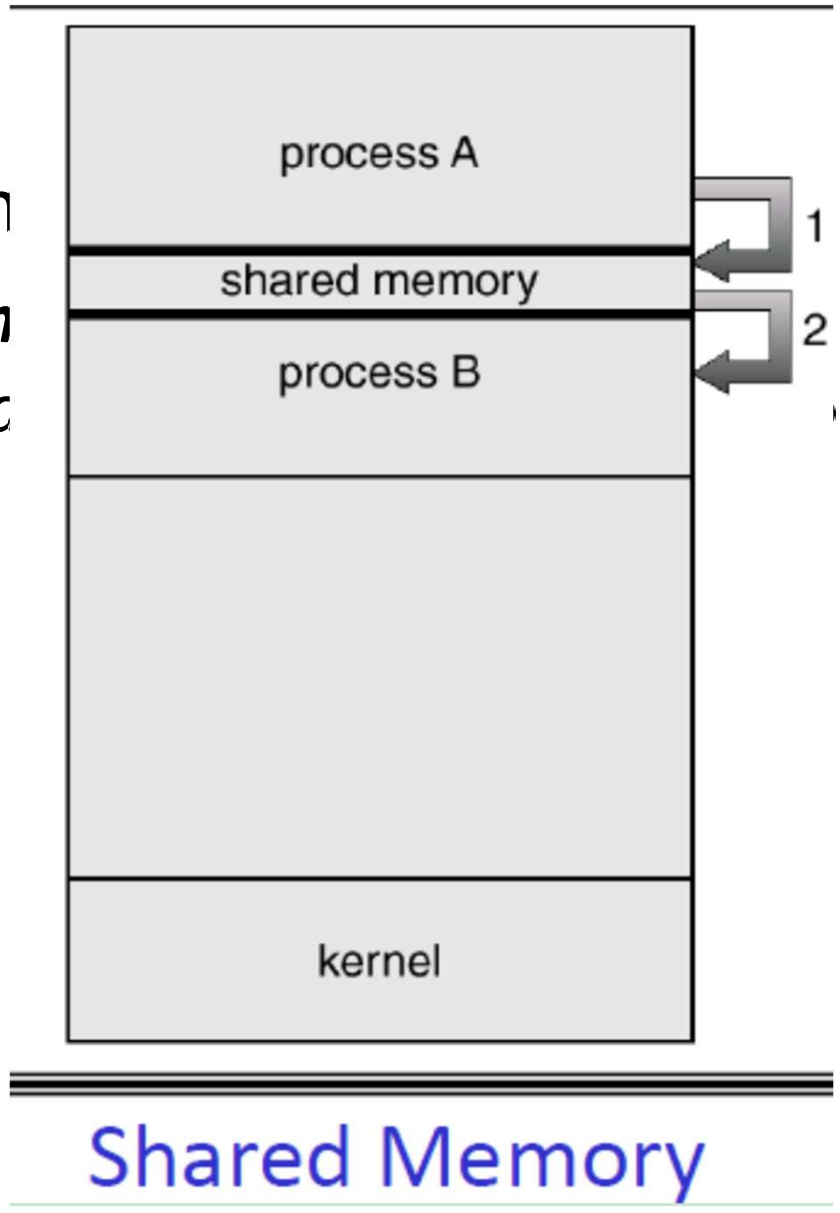
```
#include <sys/mman.h>
```

```
int munmap(void *addr, size_t length);
```

```
EX: munmap(0, 1024);
```

Shared

- Shared m
- A ***shared n*** is *attachec* to use.



memory that
r their owners

Example: sender

```c
int main(void){
        const char *name="OS";
        int shm_fd;
        char buffer[1024];
        char *shm_base;
        char *ptr;
        shm_fd=shm_open(name,O_CREAT|O_RDWR,0666);
        if(shm_fd<0){
                perror("open error:\n");
                exit(1);
        }
        ftruncate(shm_fd,1024);
        shm_base=mmap(0,1024,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd,0);
        if(shm_base==MAP_FAILED){
                perror("shm_base error:\n");
                exit(1);
        }
        fgets(buffer,sizeof(buffer),stdin);
        ptr=shm_base;
        ptr+=sprintf(ptr,"%s",buffer);
        if(munmap(shm_base,1024)==-1){
```

Example(cont.): receiver

```c
int main(void){
        const char *name="OS";
        int shm_fd;
        char buffer[1024];
        char *shm_base;
        char *ptr;
        shm_fd=shm_open(name,O_RDONLY,0666);
        if(shm_fd<0){
                perror("open error:\n");
                exit(1);
        }
        shm_base=mmap(0,1024,PROT_READ,MAP_SHARED,shm_fd,0);
        if(shm_base==MAP_FAILED){
                perror("shm_base error\n");
                exit(1);
        }
                printf("%s",shm_base);
                return 0;
}
```

Example(cont.)

```
andrew@andrew-ubuntu:~$ ./sender
hello world
andrew@andrew-ubuntu:~$ ./receiver
hello world
```

Outline

- **Creating Processes by *fork()***
- **IPC by Shared Memory**
- **Homework Assignment**

Homework 1:Create Processes and Do the Matrix Computation

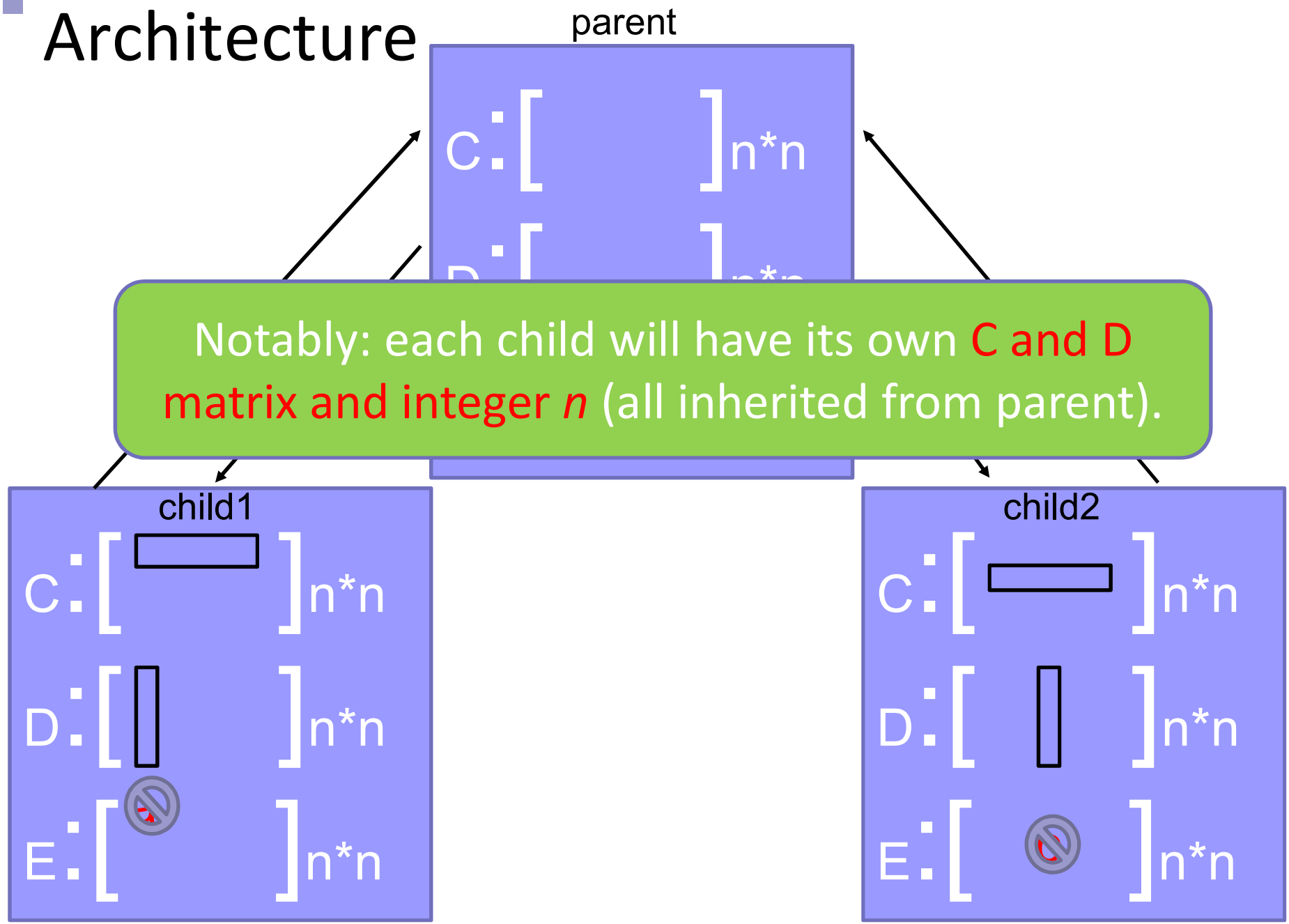
- Basic(70pt): calling *fork()* to perform matrix computation without IPC shared memory.

- Advanced(30pt): calling *fork()* to perform matrix computation through IPC shared memory

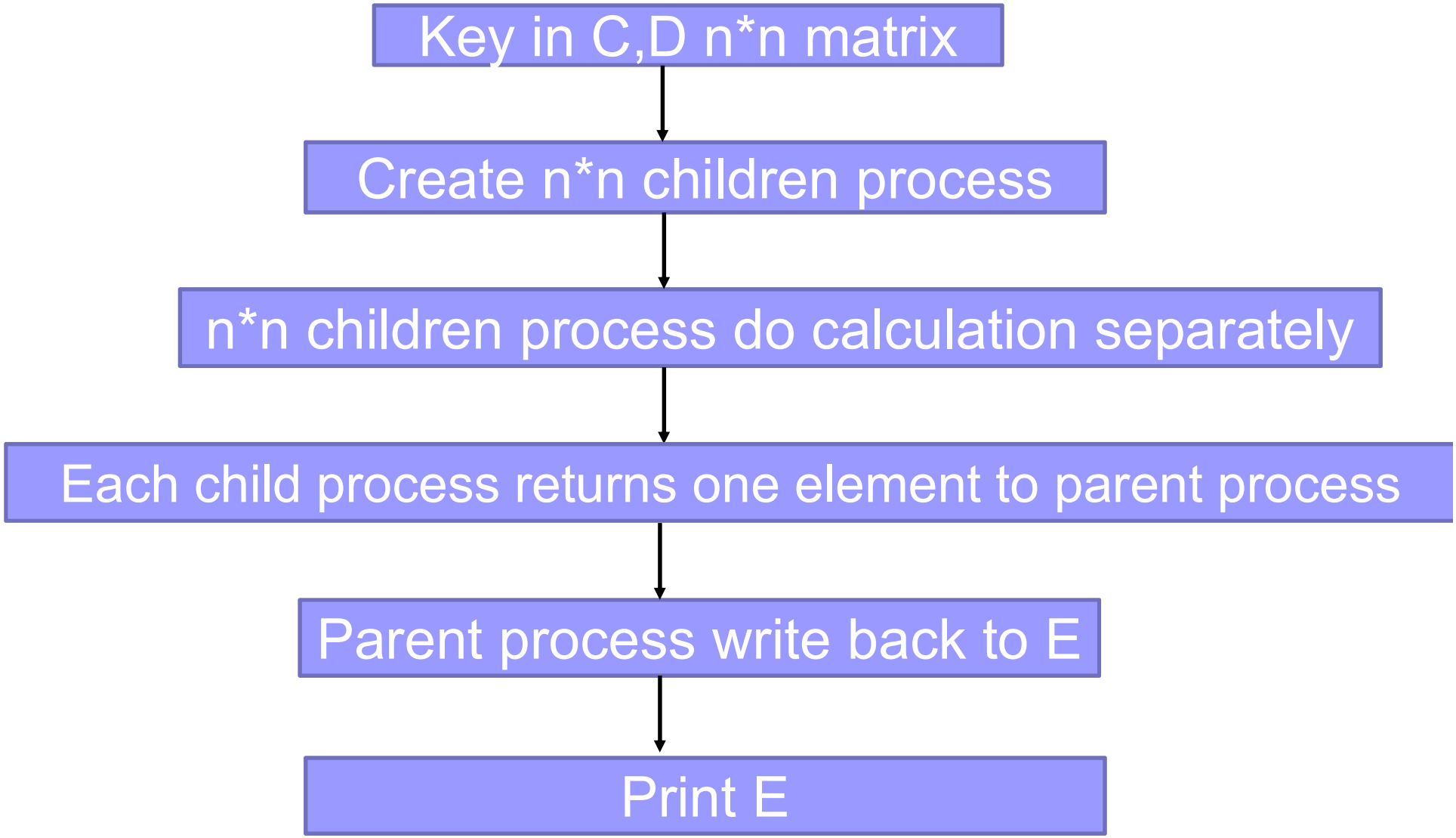
Basic(70pt)

1. Key in an integers, *n*, as matrix size
2. Key in two *n**n matrix C,D
    1. For matrix C,D, you can key in by yourself. But each element needs to be an integer.
3. Perform matrix computation: C*D = E (output)
    1. EX: if n=3, then fork 3*3=9 children process
    2. Each child process response for one element computation.
    3. Then, each child process return the result by passing it as the parameter in the *exit()* function
    4. Finally, parent process repeatedly calls *wait()* to obtain the result passed by all children processes

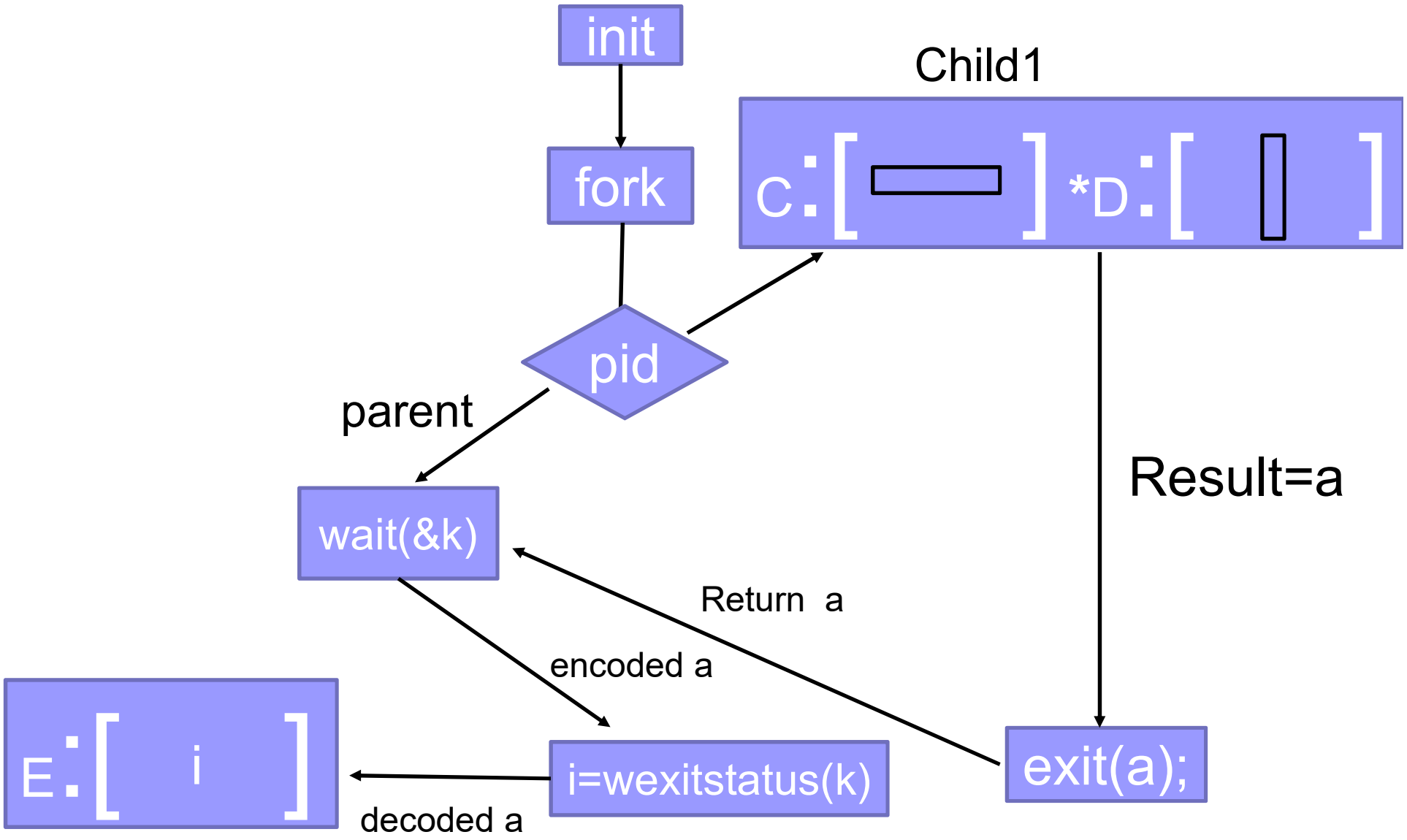
Architecture



Flow char(1/2)


Key in C,D n*n matrix
Create n*n children process
n*n children process do calculation separately
Each child process returns one element to parent process
Parent process write back to E
Print E

Flow char(2/2)


init
fork
pid
parent
wait(&k)
Child1
C:[ ] *D:[ ]
Result=a
Return a
encoded a
E:[i]
i=wexitstatus(k)
exit(a);
decoded a

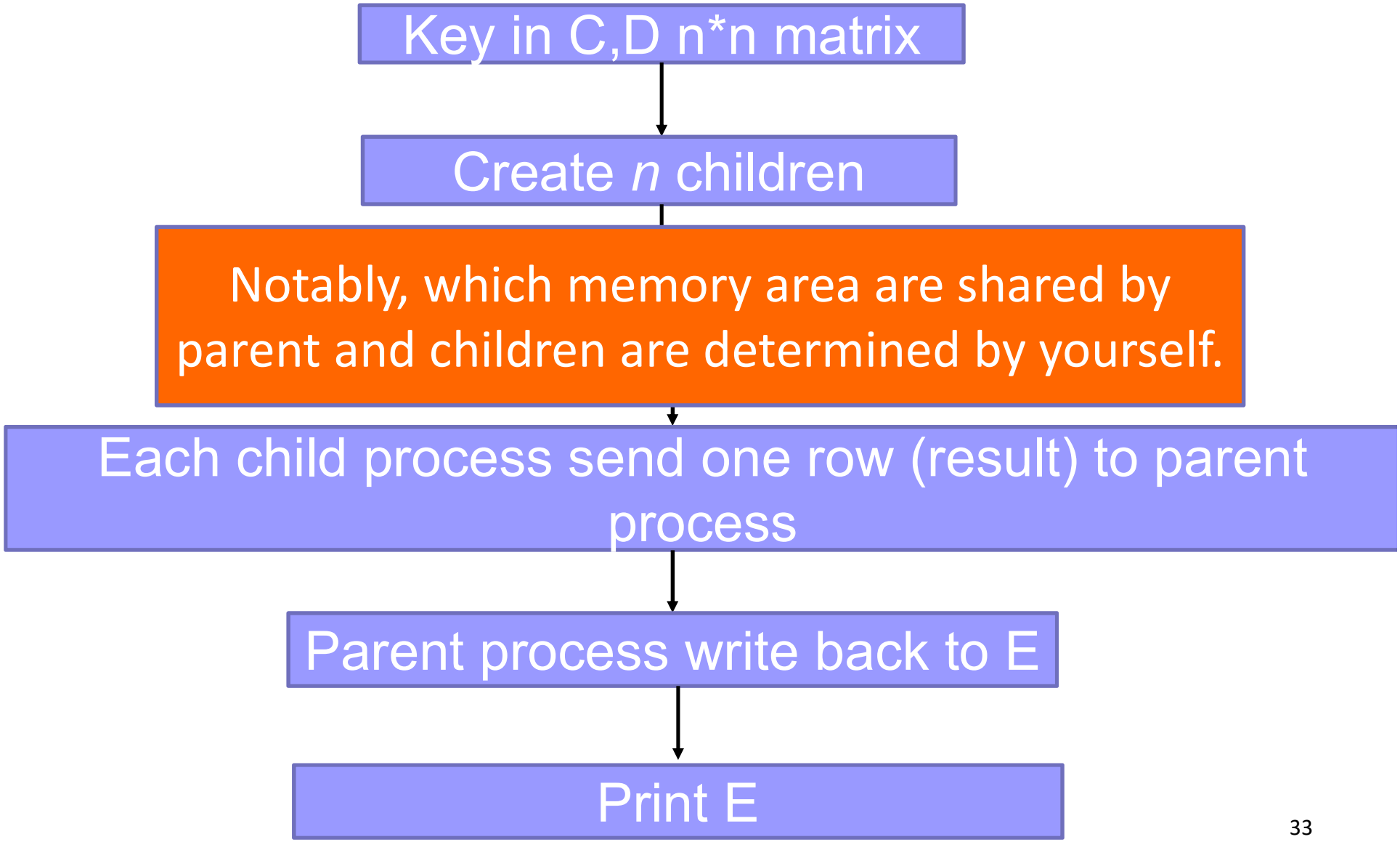
result

```
matrix size:
3*3
matrix c
1 2 3
4 5 6
7 8 9
matrix d
9 8 7
6 5 4
3 2 1
result:
30 24 18
84 69 54
138 114 90
```

Advance (30pt)

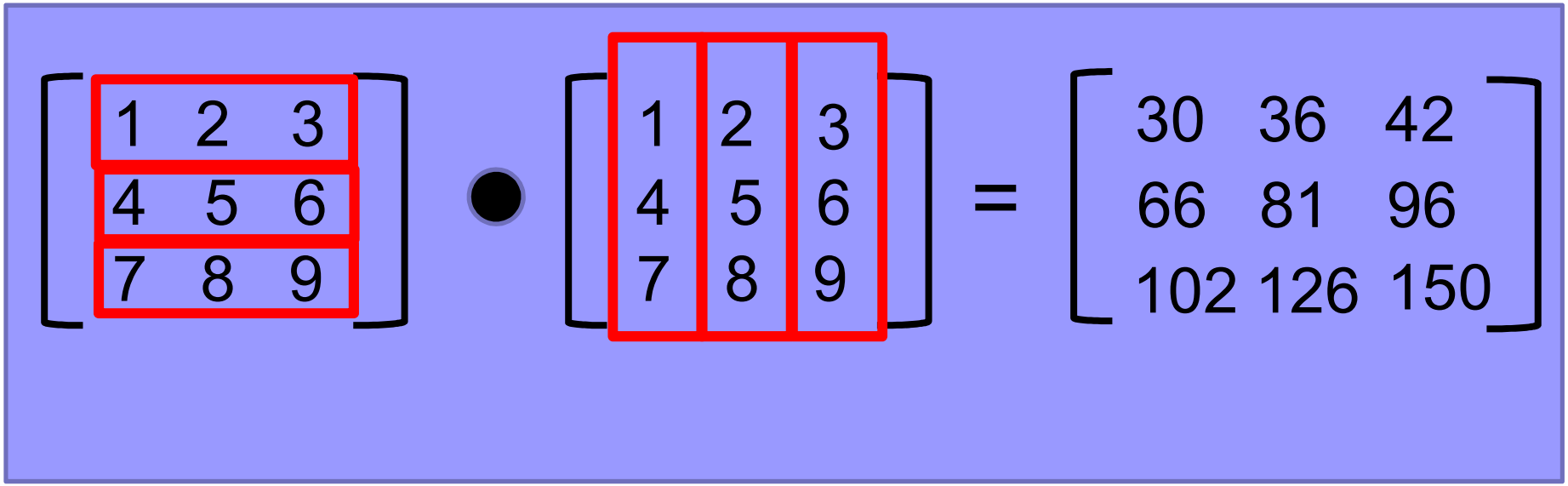
1. Key in an integer *n* as matrix size
2. Key in two *n***n* matrix C,D
3. Perform matrix computation : C*D = E (output)
    1. Parent process (main process) fork ***n*** child process (total *n*+1 process)
    2. The *i*-th child process drives the *i*-th row of ***E*** by multiplying *i*-th row of C with D
    3. Once *i*-th child process completes the calculation, it sends the result, i.e., *i*-th row of ***E***, back to the parent.
    4. After all children complete the calculation, parent process print the complete output matrix ***E***

Flow char


Key in C,D n*n matrix
Create *n* children
Notably, which memory area are shared by parent and children are determined by yourself.
Each child process send one row (result) to parent process
Parent process write back to E
Print E

Example

child3

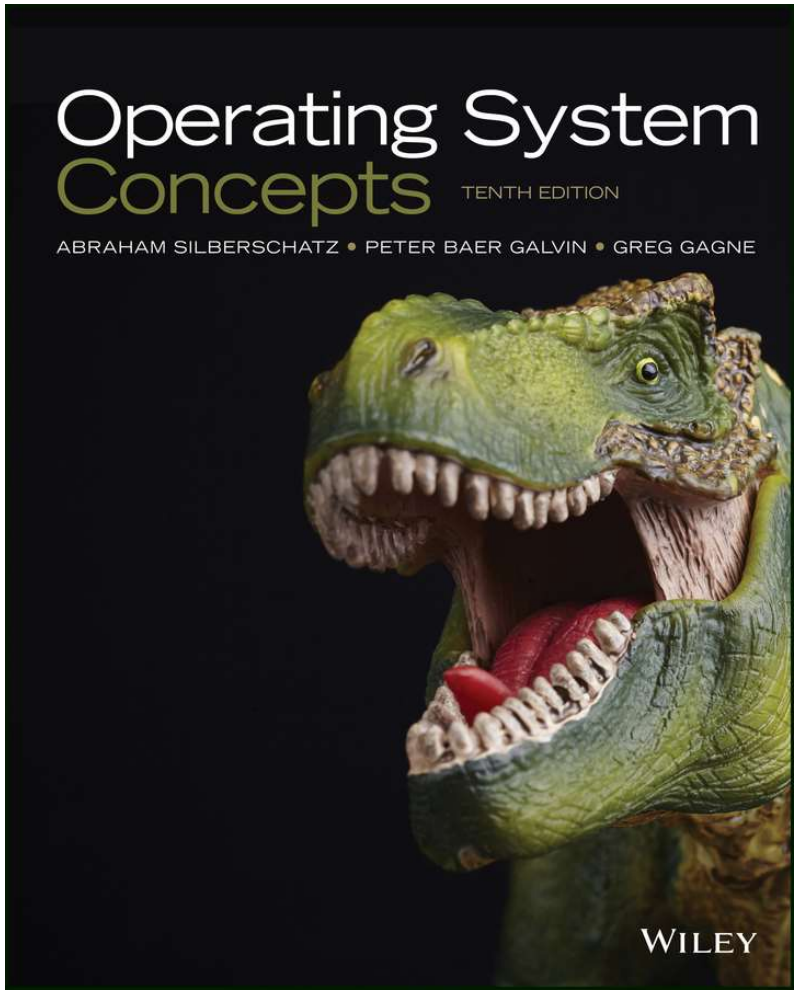


Result

```
matrix size:
3*3
matrix c
1 2 3
4 5 6
7 8 9
matrix d
11 12 13
14 15 16
17 18 19
result:
90 96 102
216 231 246
342 366 390
```

Reference

- Operating System Concepts, 10th Edition



Turn in

- Deadline

  2020/11/19 PM.11:59:59
- Upload to iLearning
- File name
  - HW1_ID.zip (e.g. HW1_4106056000.zip)
    - Source code
      - .c file
    - Word(explain your code and post a screenshot of the result)
- If you don’t hand in your homework on time, your score will be deducted 10 points every day.

TA

- Name：Sheng-Ying Huang
- Email：g109056252@mail.nchu.edu.tw
- Lab：OSLab (1001)