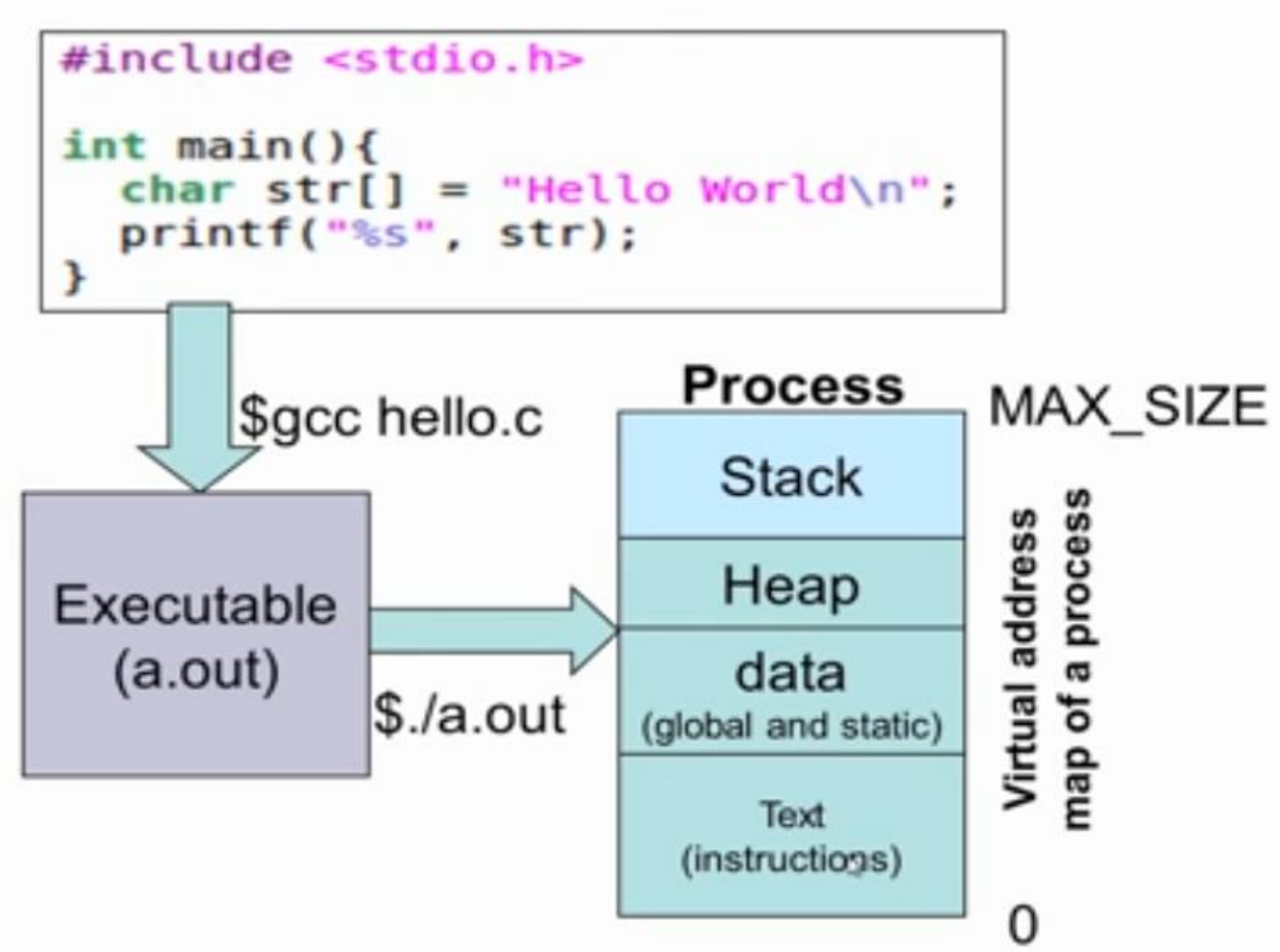
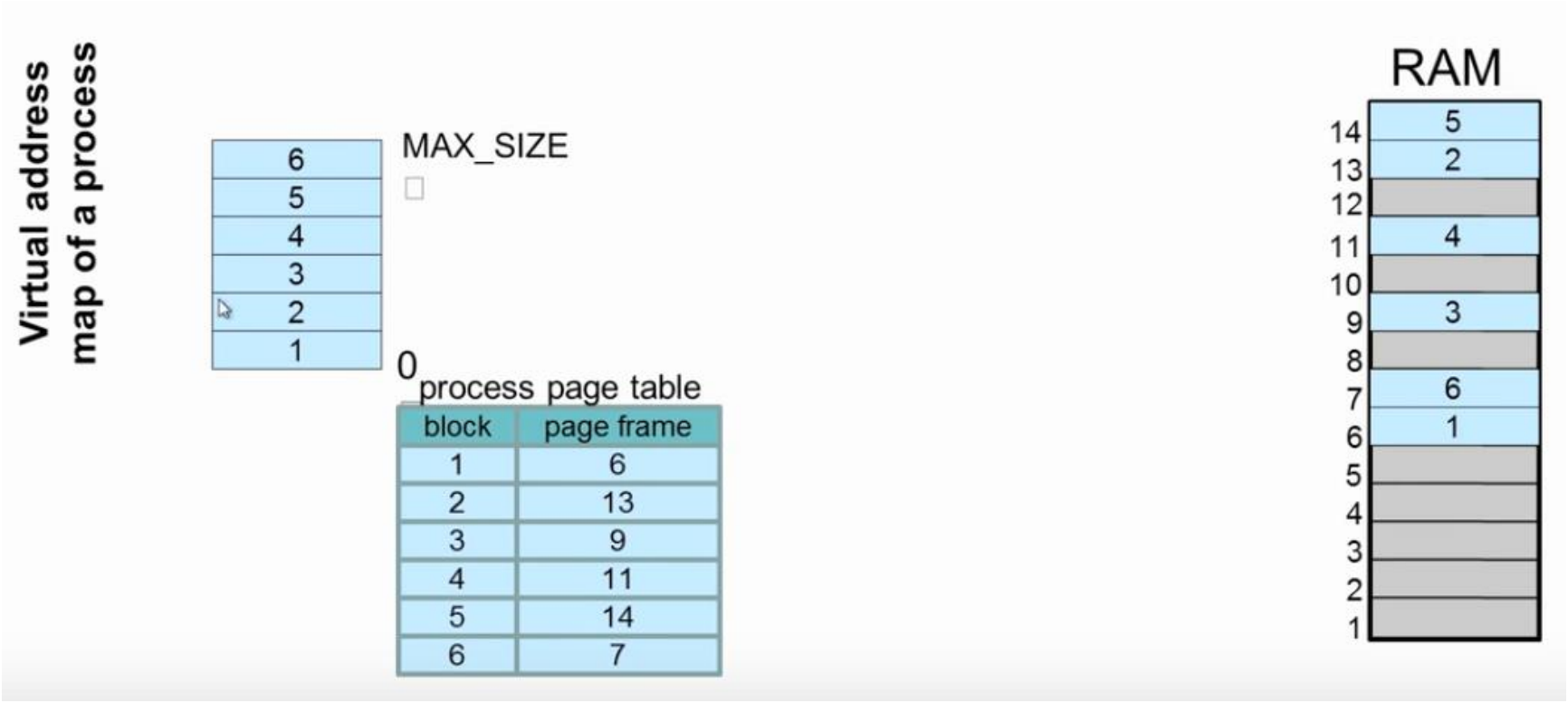
Process_And_Thread


#include <stdio.h>
int main(){
char str[] = "Hello World\n";
printf("%s", str);
}
$gcc hello.c
Executable
(a.out)
$./a.out
Process
Stack
Heap
data
(global and static)
Text
(instructions)
MAX_SIZE
Virtual address
map of a process
0

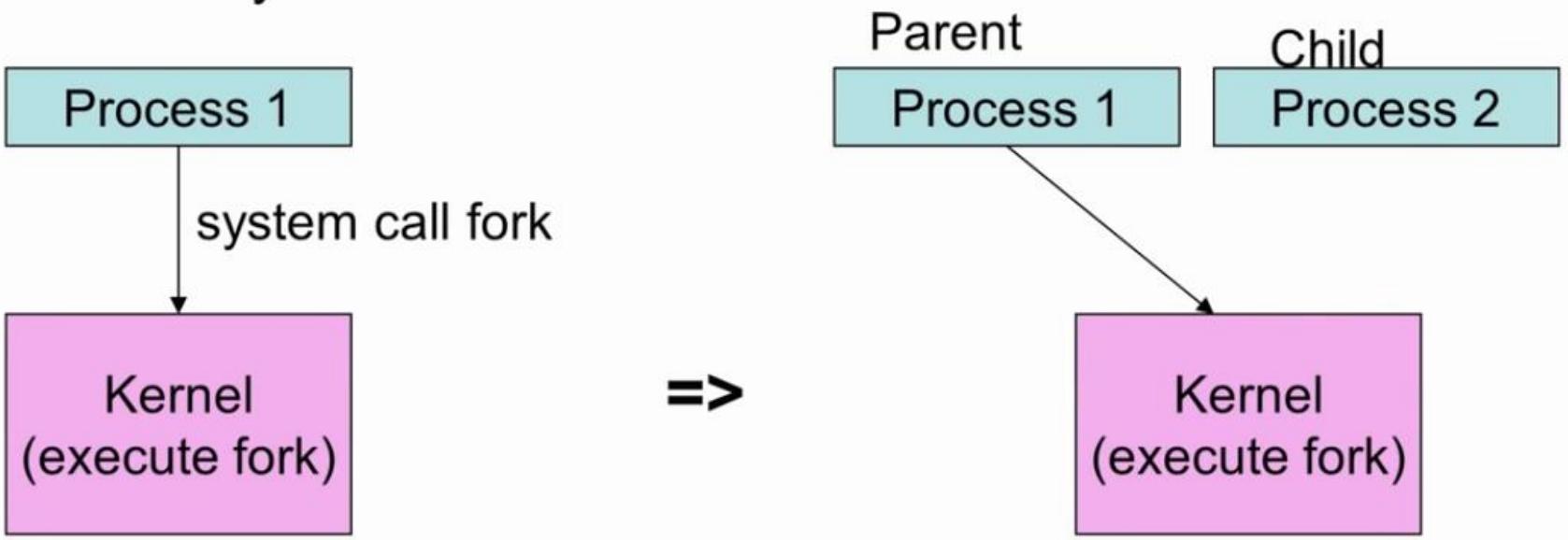
Virtual Address Map of a Process



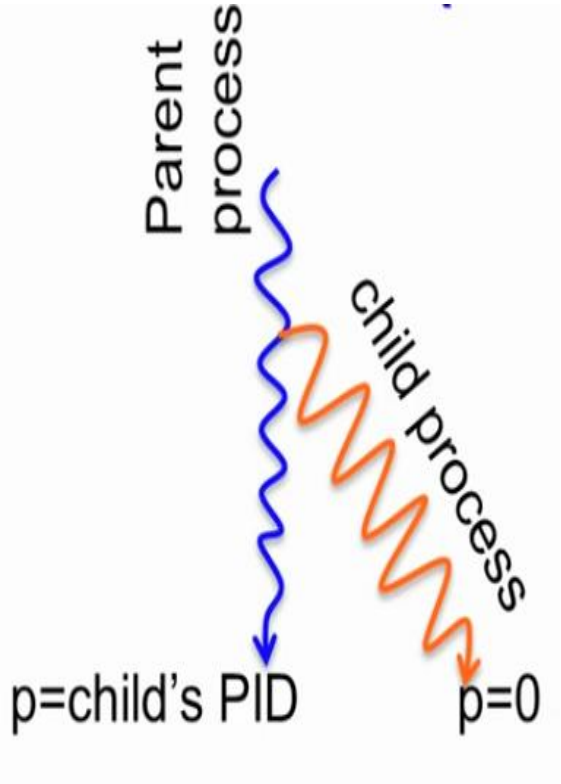
Creating a Process

Cloning

- Child process is an exact replica of the parent
- Fork system call

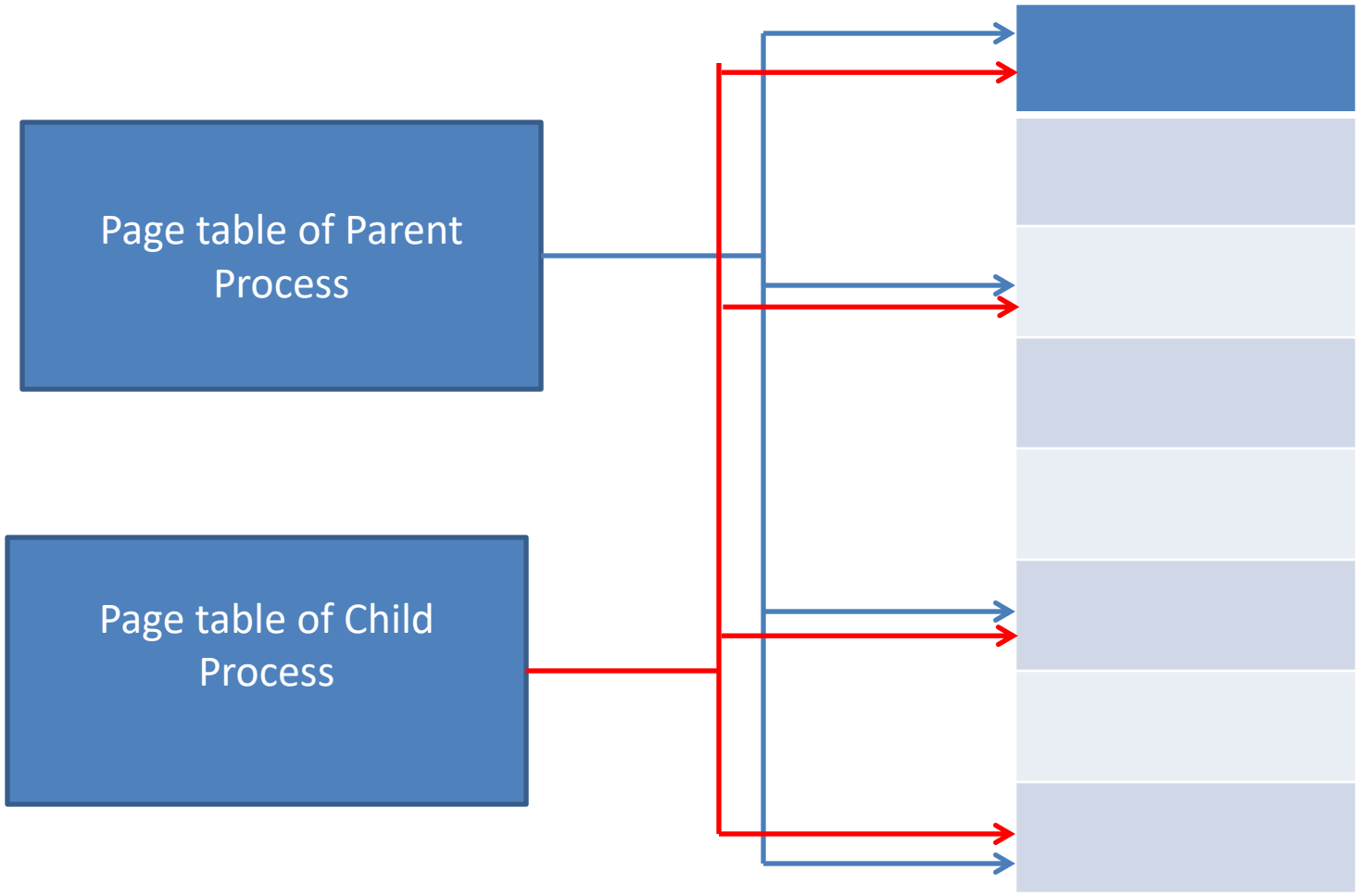


Creating Child Process Using fork() system call




```
int p;

p = fork();
if (p > 0){
   printf("Parent : child PID = %d", p);
   p = wait();
   printf("Parent : child %d exited\n", p);
} else if (p == 0){
  printf("In child process");
  exit(0);
} else{
   printf("Error\n ");
}
```

Page Table Duplicated in child process



New PCB created for Child Process

- Find new PID
- Set state to New
- Set Pointers to newly formed page-table
- Copy information like files opened, current working directory from PCB of parent process

Example

```
int i=23, pid;
pid = fork();
if (pid > 0){
        sleep(1);
        printf("parent : %d\n", i);
        wait();
} else{

        printf("child : %d\n", i);
}
```

Output

```
child  : 23
Parent : 23
```

Example

```
int i=23, pid;
pid = fork();
if (pid > 0){
        sleep(1);
        printf("parent : %d\n", i);
        wait();
} else{
        i = i + 1;
        printf("child : %d\n", i);
}
```

Output

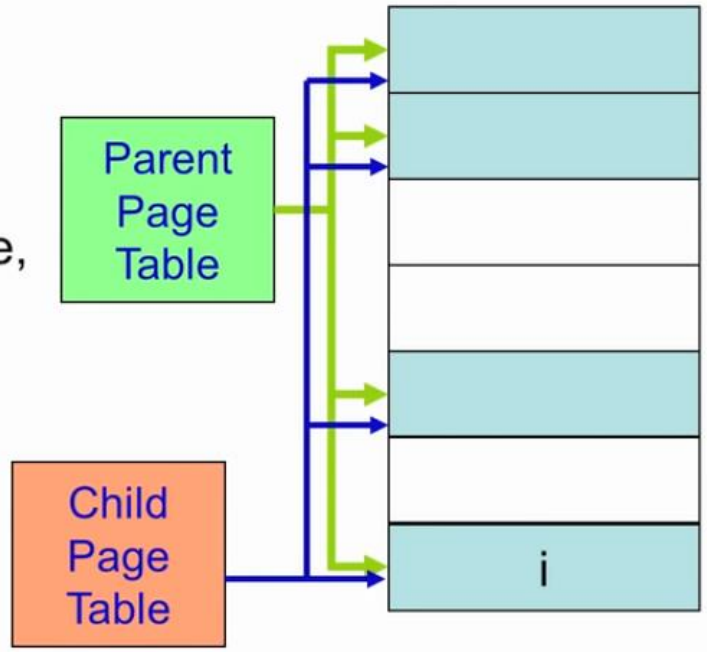
```
child  : 24
Parent : 23
```

Copy on Write

All parent pages initially marked as shared
(a bit in parent and child page tables)

When data in any of the shared pages change,
OS intercepts and makes a copy of the page.

Thus, parent and child will have different
copies of this page
(all other pages remain the same)

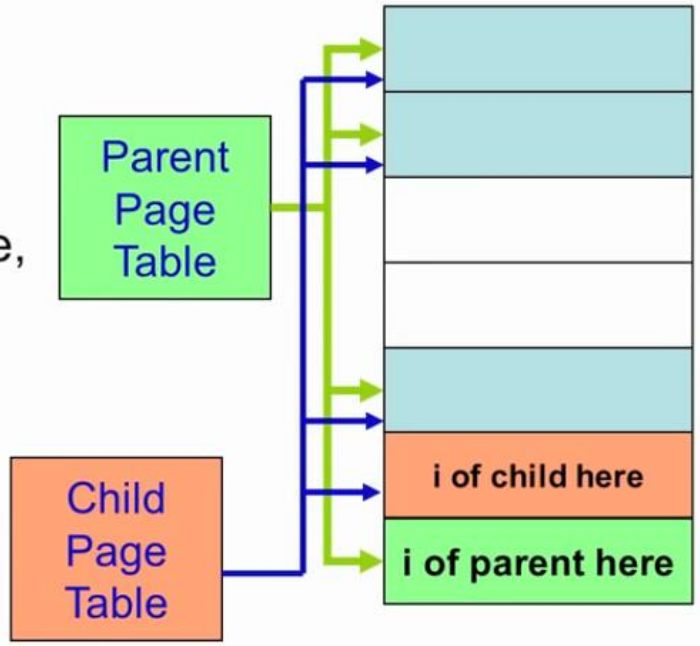


Copy on Write

All parent pages initially marked as shared
(a bit in parent and child page tables)

When data in any of the shared pages change,
OS intercepts and makes a copy of the page.

Thus, parent and child will have different
copies of this page
(all other pages remain the same)



Execution of Completely new process

Two step process

First fork and then exec

exec will load the executable /usr/bin/ls in main memory and then execute.

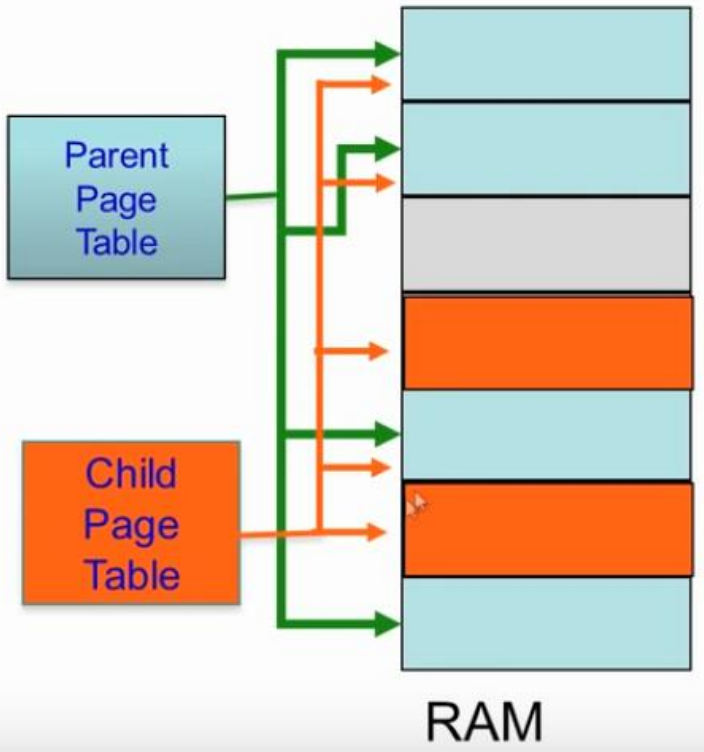
```
pid_t pid;
int i=23;
pid=fork();

If (pid >0) {
printf("In parent:i=%d\n",i);
}

if(pid == 0){
execlp("/usr/bin/ls","",NULL);
exit(0);
}
```

fork and exec

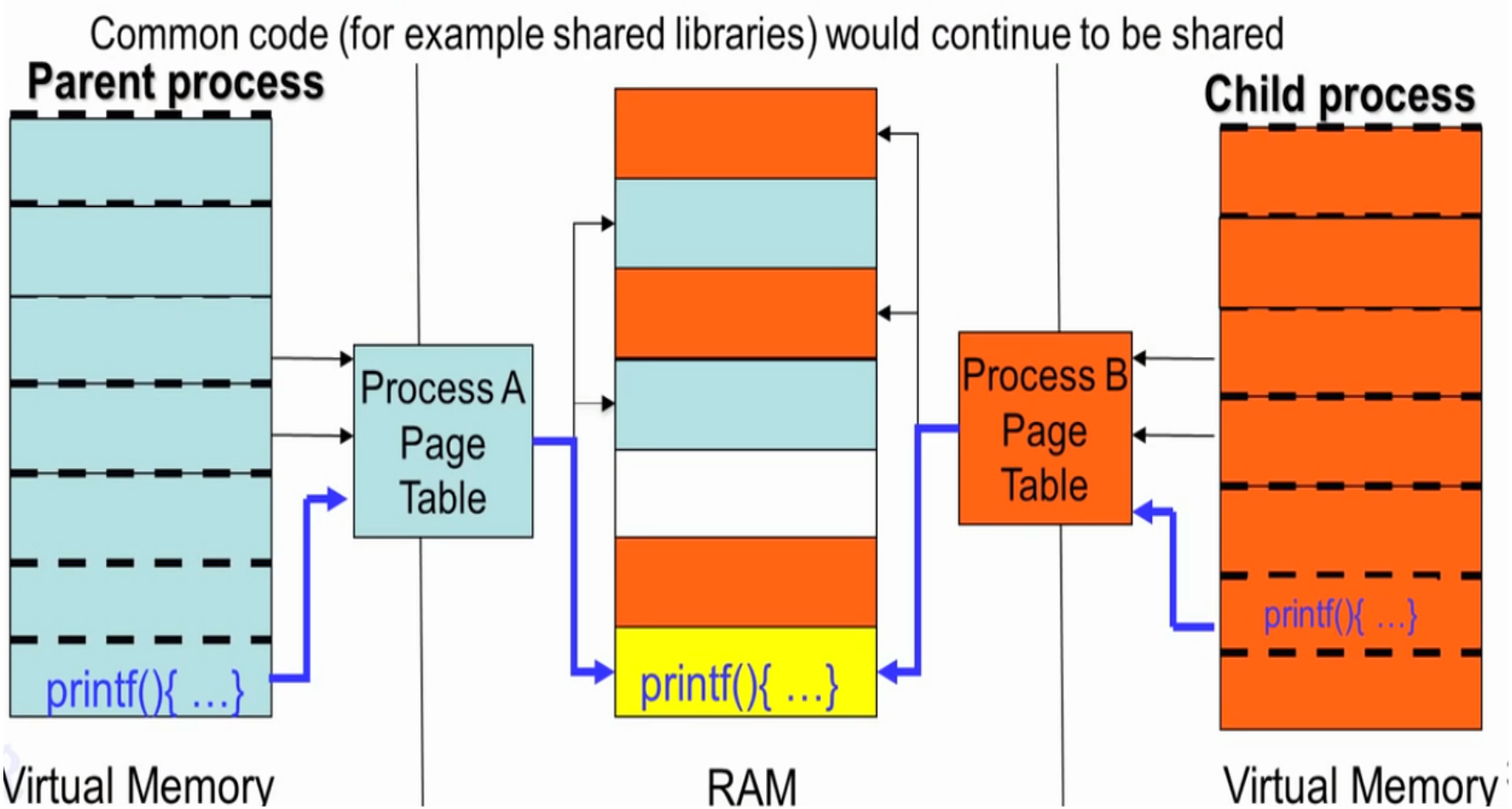
fork would create a child which is an exact replica of the parent

In the child process, *exec* would load blocks from the hard disk into page frames on demand.

– The child page tables would be updated as required



Advantage of CoW



Process Termination by Exit

exit()

- Called in child process
- Results in the process terminating
- The return status (0 here), is passed on to the parent.

(this is a voluntary termination)

```
int pid;

pid = fork();
if (pid > 0){
   pid = wait();
} else{
  execlp("./a.out", "", NULL);
  exit(0);
```

Involuntary Termination

Involuntary : kill(pid, signal)

- Signal can be sent by another process or by OS
- pid is for the process to be killed

wait system call

wait()

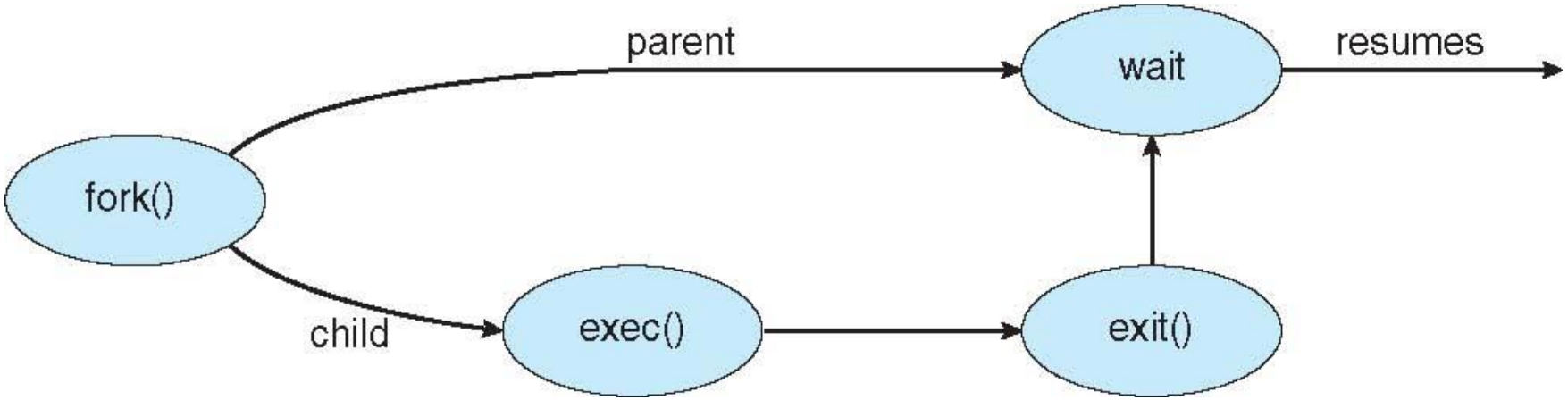
- Called in parent
- Parent goes to block state
  - Until one of its children exits
  - If no children executing, then -1 is returned

```
int pid;

pid = fork();
if (pid > 0){
   pid = wait();
} else{
   execlp("./a.out", "", NULL);
   exit(0);
```

More Example

```
pid=fork();
pid=fork();

if(pid == 0){
printf("In child\n");
exit(0);
}

else{
printf("Parent:pid before wait=%d\n",pid);
pid=wait();
printf("Parent:pid after wait=%d\n",pid);
}
```

```
[sourav@pg ~]$ ./a.out
Parent:pid before wait=26227
In child
In child
Parent:pid before wait=26228
Parent:pid after wait=26227
Parent:pid after wait=26228
```


parent
wait
resumes
fork()
child
exec()
exit()

Zombie

When a process terminates it becomes a zombie (or defunct process)

- PCB in OS still exists even though program is no longer executing
- Why? So that the parent process can read the child's exit status (through wait system call)

When parent reads status,

- zombie entries removed from OS… process reaped!

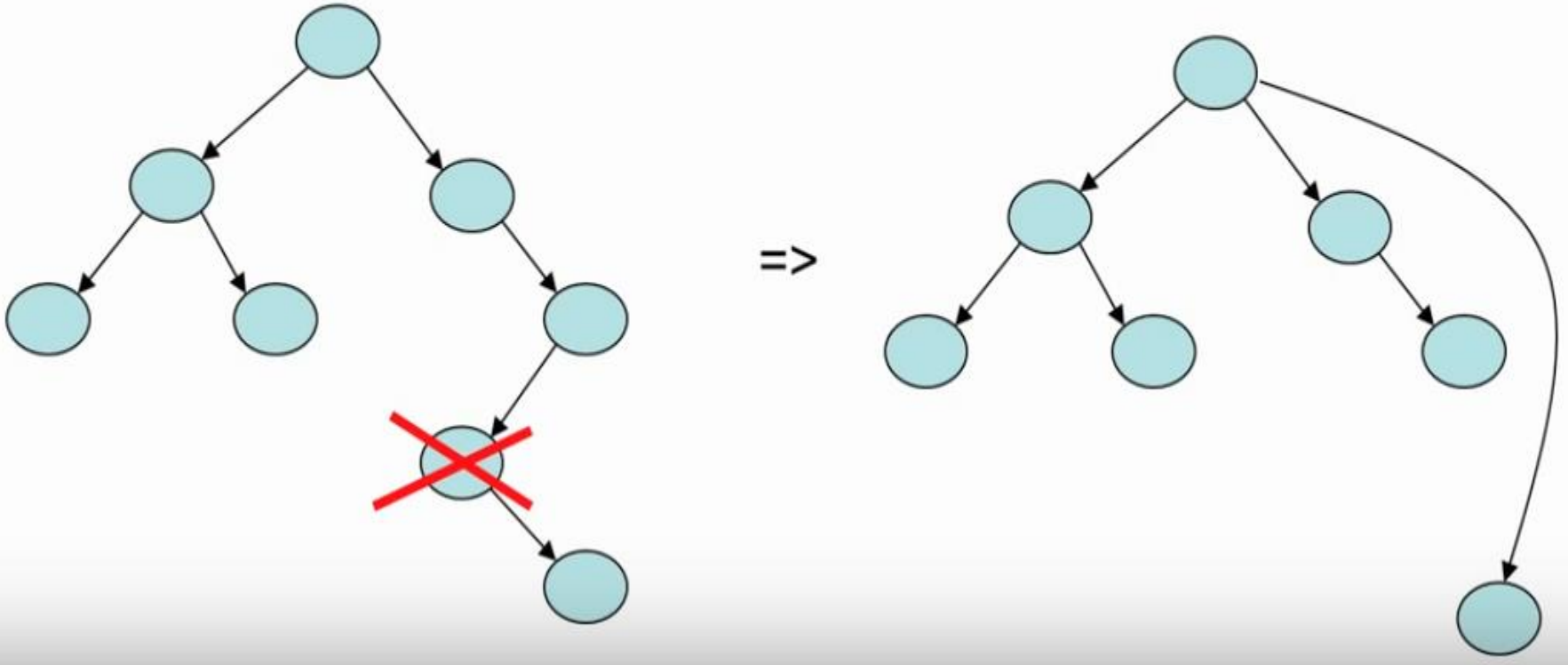
Suppose parent does'nt read status

- Zombie will continue to exist infinitely … a resource leak
- These are typically found by a reaper process

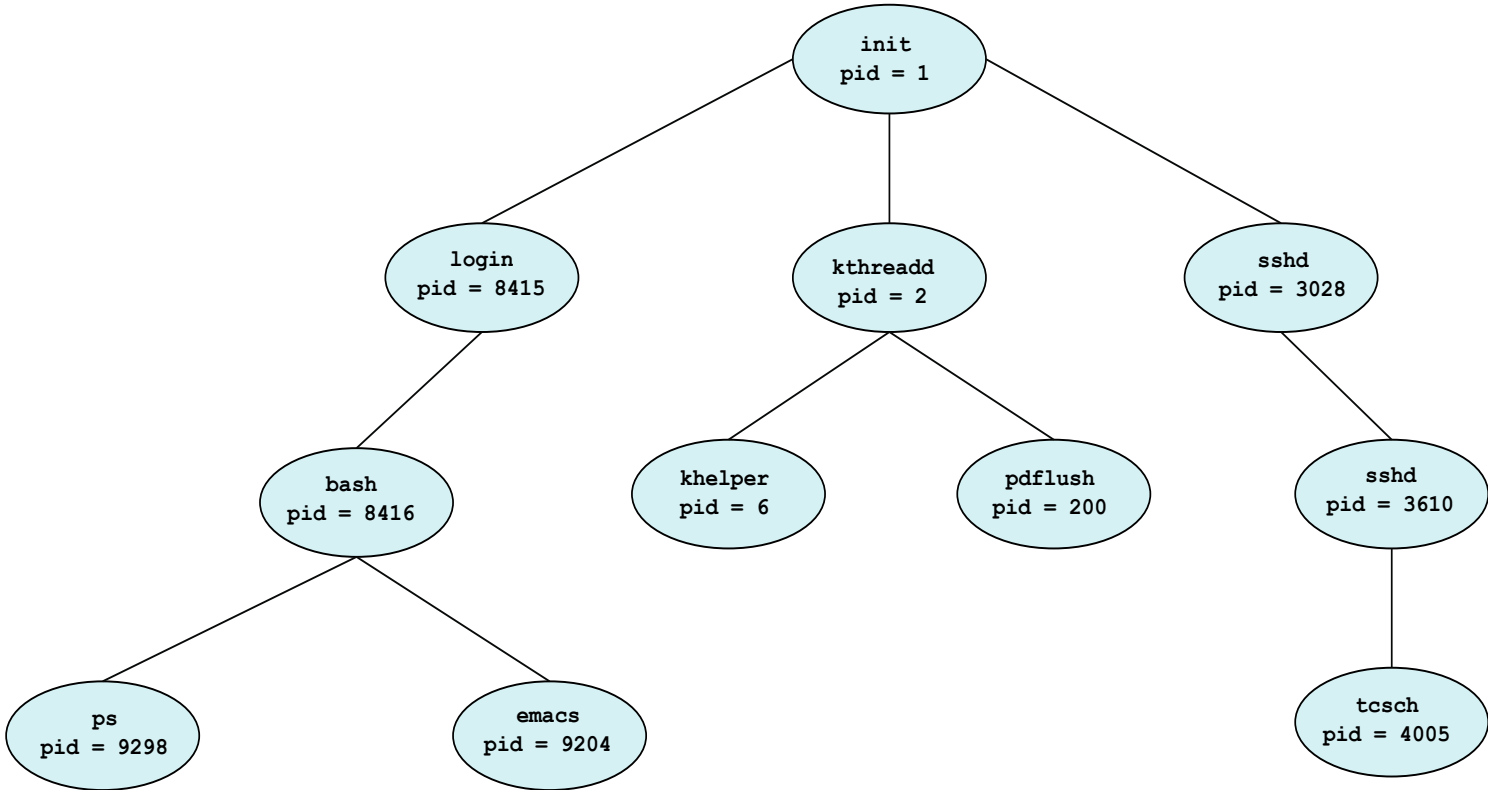
Orphan

When a parent process terminates before its child

Adopted by first process (/sbin/init)



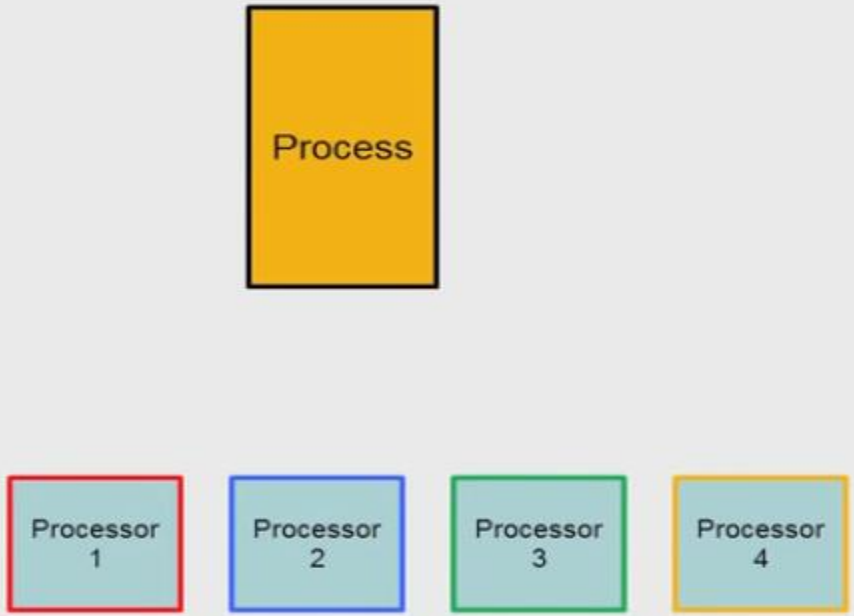
A Tree of Processes in Linux



Thread

Example

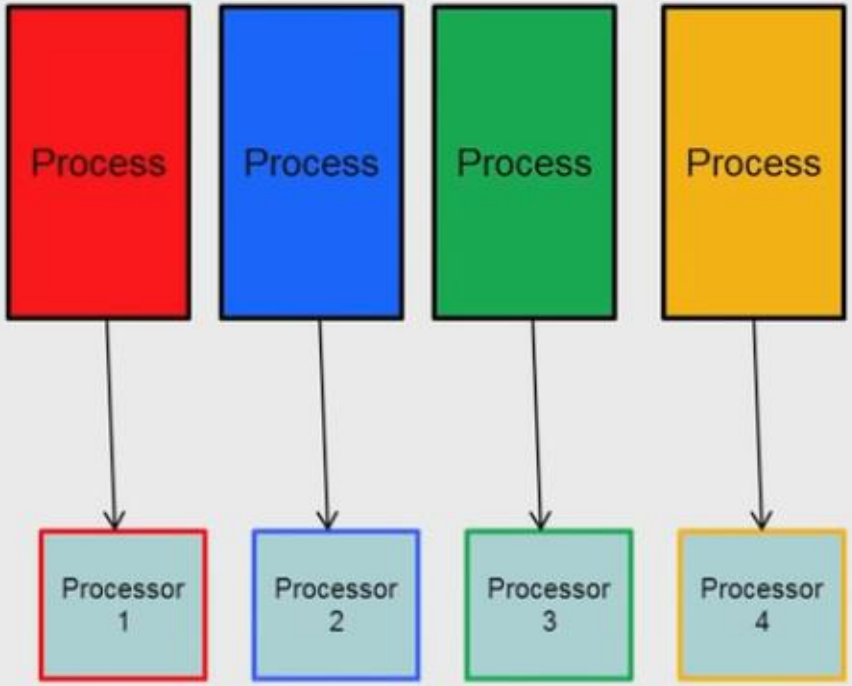
```c
#include <stdio.h>


unsigned long addall(){
  int i=0;
  unsigned long sum=0;

  while (i< 10000000){
     sum += i;
     i++;
  }
  return sum;
}


int main()
{
   unsigned long sum;
   srandom(time(NULL));
   sum = addall();
   printf("%lu\n", sum);
}
```



Speeding Up

10000000 / 4 = 2500000

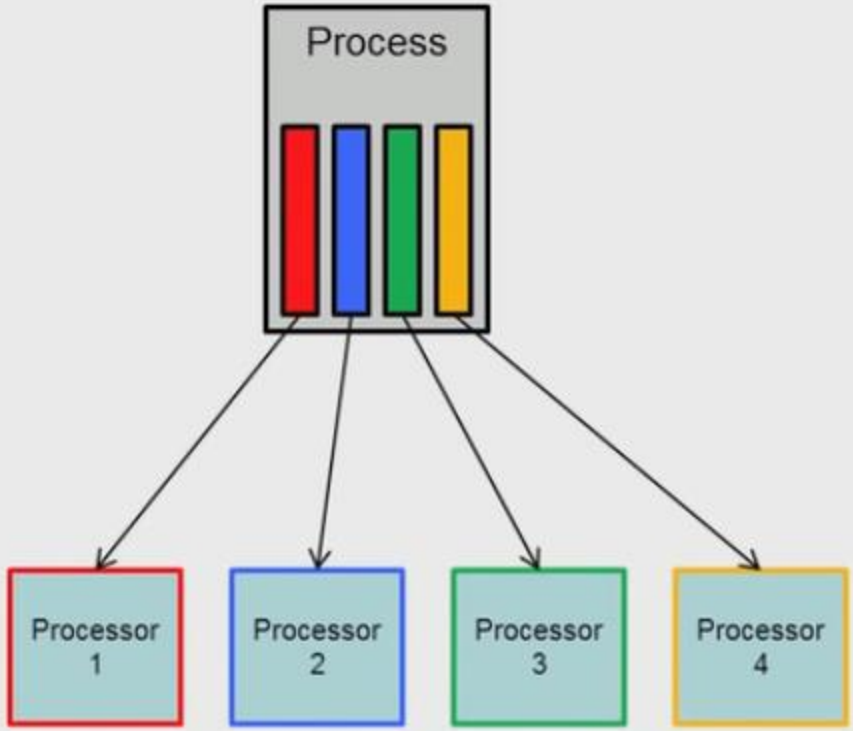
Create 4 processes, each loop does 1/4th of the work



Thread Model

10000000 / 4 = 2500000

Create 1 process with 4 threads; each loop does 1/4th of the work



Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

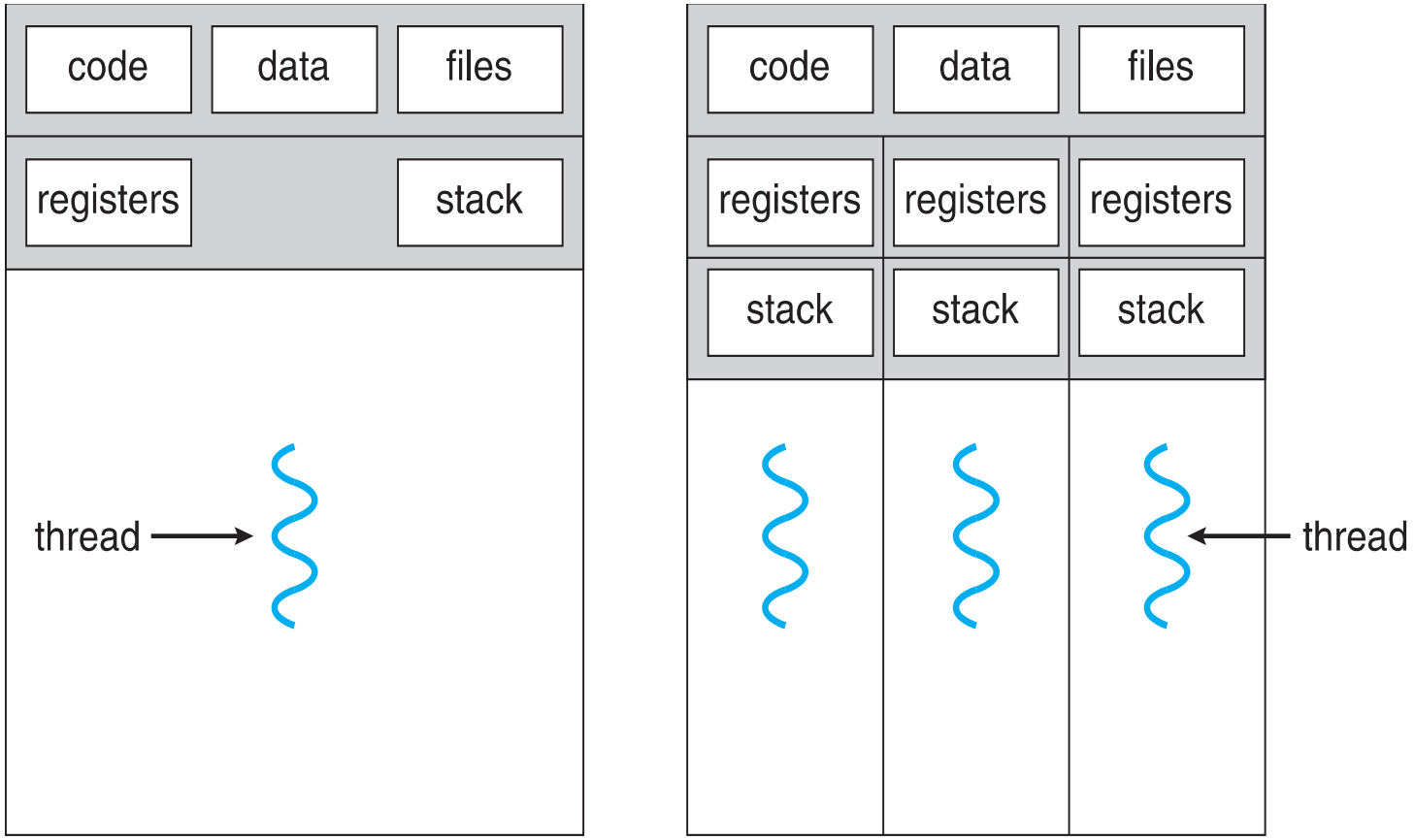
Comparison of Process and Thread

Intel 2.6GHz Xeon E5-2670 (16 core)	9
Intel 2.8GHz Xeon E5660 (12 core)	6.3
AMD 2.3 GHz Opteron (16 cores)	10.5
AMD 2.4 GHz Opteron (8 cores)	12.5
IBM 4.0 GHz Power6 (8 cores)	6

(Time for creation 50000 process/threads)

Moreover Context switching is 30 times faster in thread

Single and Multithreaded Processes



single-threaded process

multithreaded process

Management of Thread

Two strategies

- User threads
  - Thread management done by user level thread library. Kernel knows nothing about the threads.
- Kernel threads
  - Threads directly supported by the kernel.
  - Known as light weight processes.

User Level Thread

Advantages:

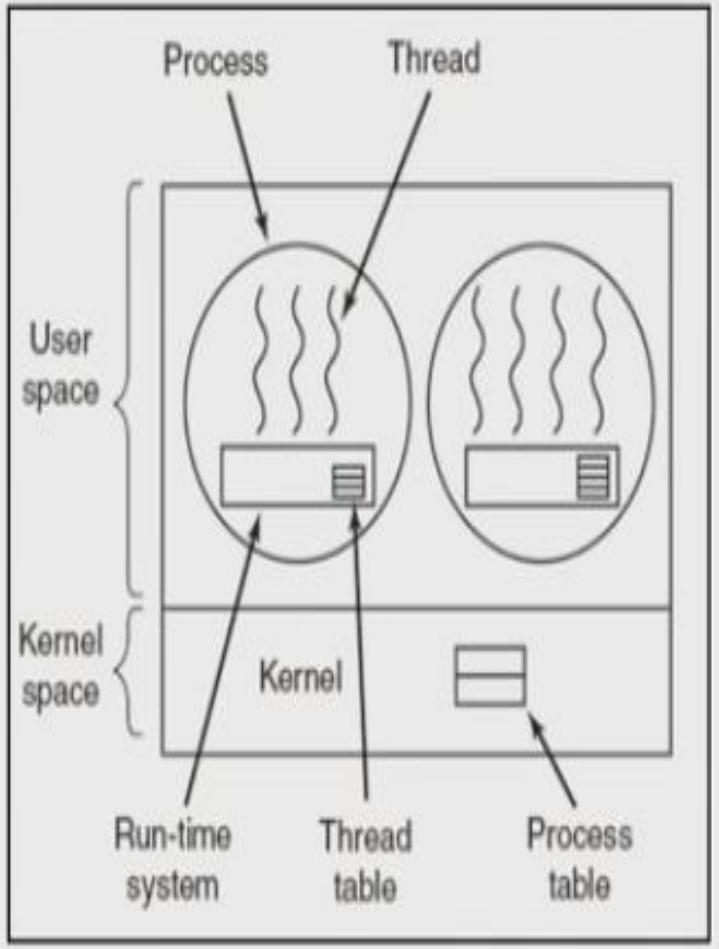
- Fast (really lightweight)

(no system call to manage threads. The thread library does everything).

- Can be implemented in an OS that does not support threading.
- Switching is fast. No switch from user to protected mode.

Disadvantages:

- Scheduling can be an issue. (Consider, one thread that is blocked on an IO and another runnable.)
- Lack of coordination between kernel and threads. (A process with 100 threads competes for a timeslice with a process having just 1 thread.)
- Requires non-blocking system calls. (If one thread invokes a system call, all threads need to wait)



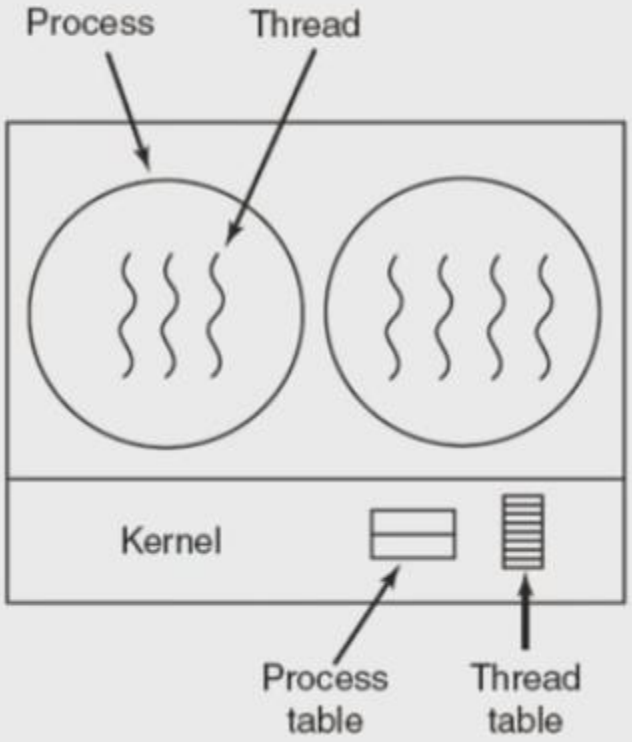
Kernel Level Thread

Advantages:

- Scheduler can decide to give more time to a process having large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

- The kernel-level threads are slow (they involve kernel invocations.)
- Overheads in the kernel. (Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads.)



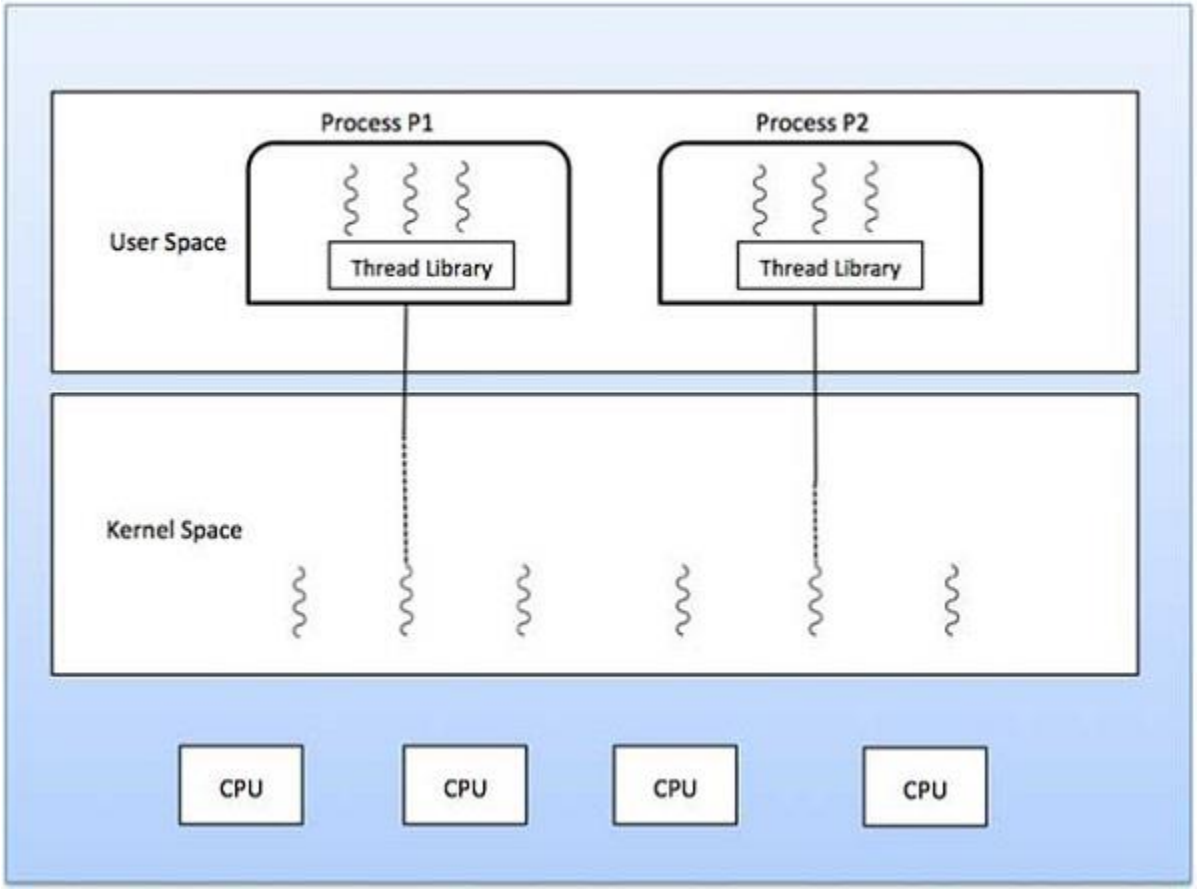
Combined Approach

- (https://www.tutorialspoint.com/operating_system/os_multi_threading.htm)
- Some operating system provide a combined user level thread and Kernel level thread facility.
- Solaris is a good example of this combined approach.
- In a combined system, multiple threads within the same application can run in parallel on multiple processors.

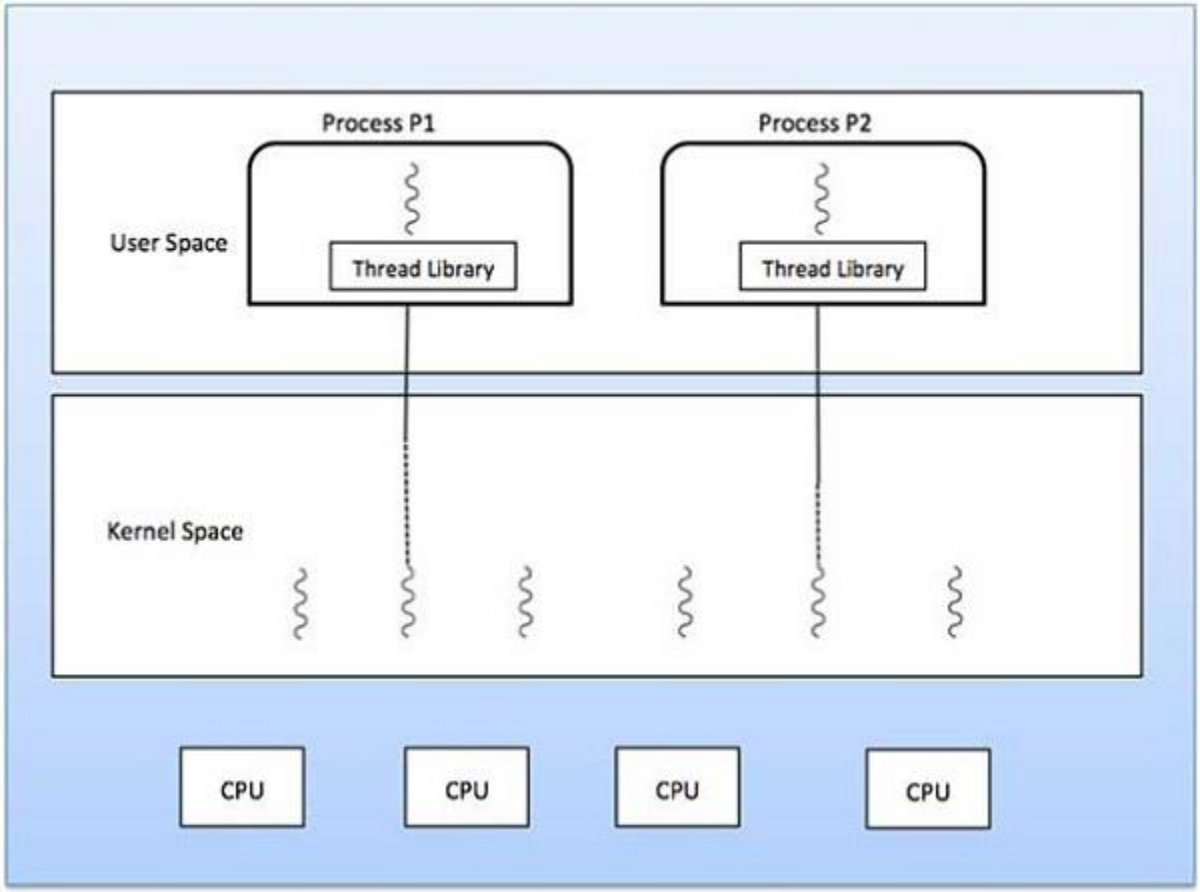
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

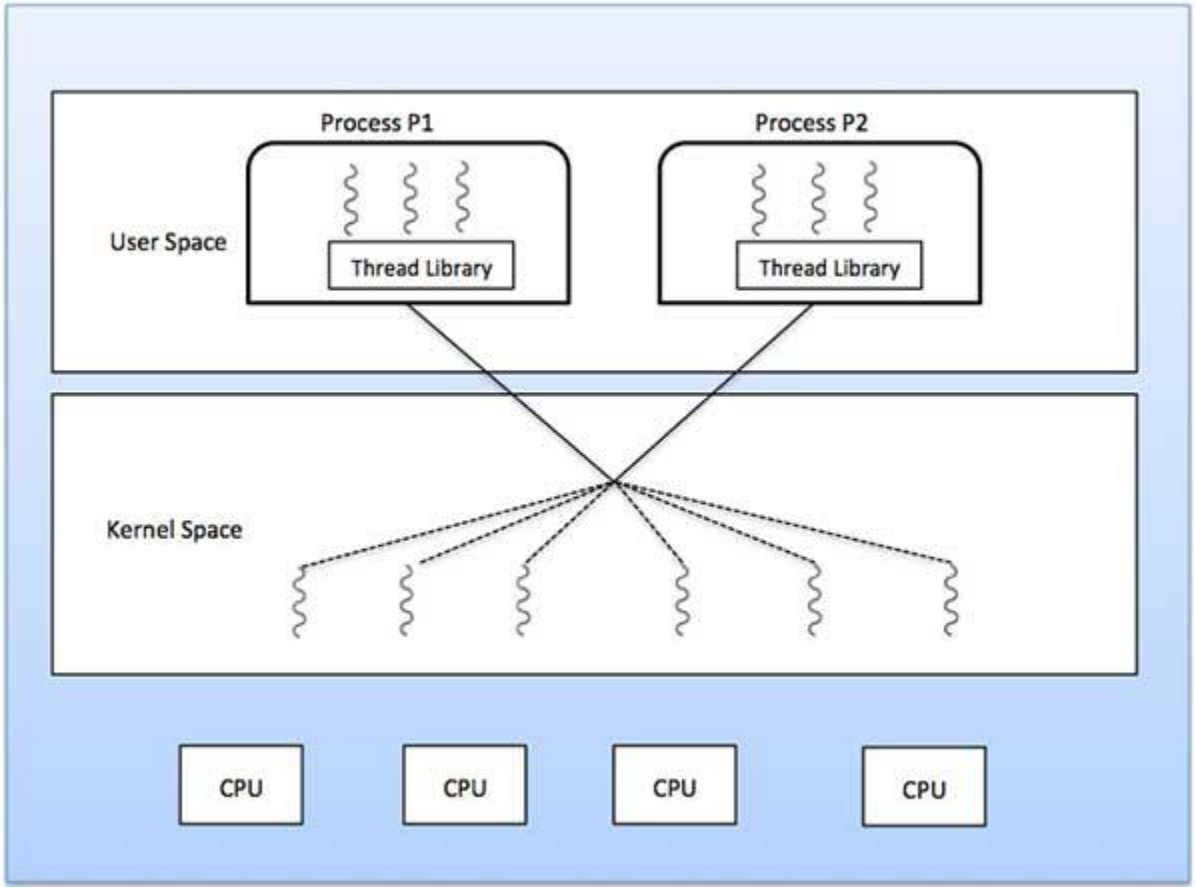
Many-to-one


Process P1
Process P2
User Space
Thread Library
Thread Library
Kernel Space
CPU
CPU
CPU
CPU

One-to-one


Process P1
Process P2
User Space
Thread Library
Thread Library
Kernel Space
CPU
CPU
CPU
CPU

Many-to-many


Process P1
Process P2
User Space
Thread Library
Thread Library
Kernel Space
CPU
CPU
CPU
CPU

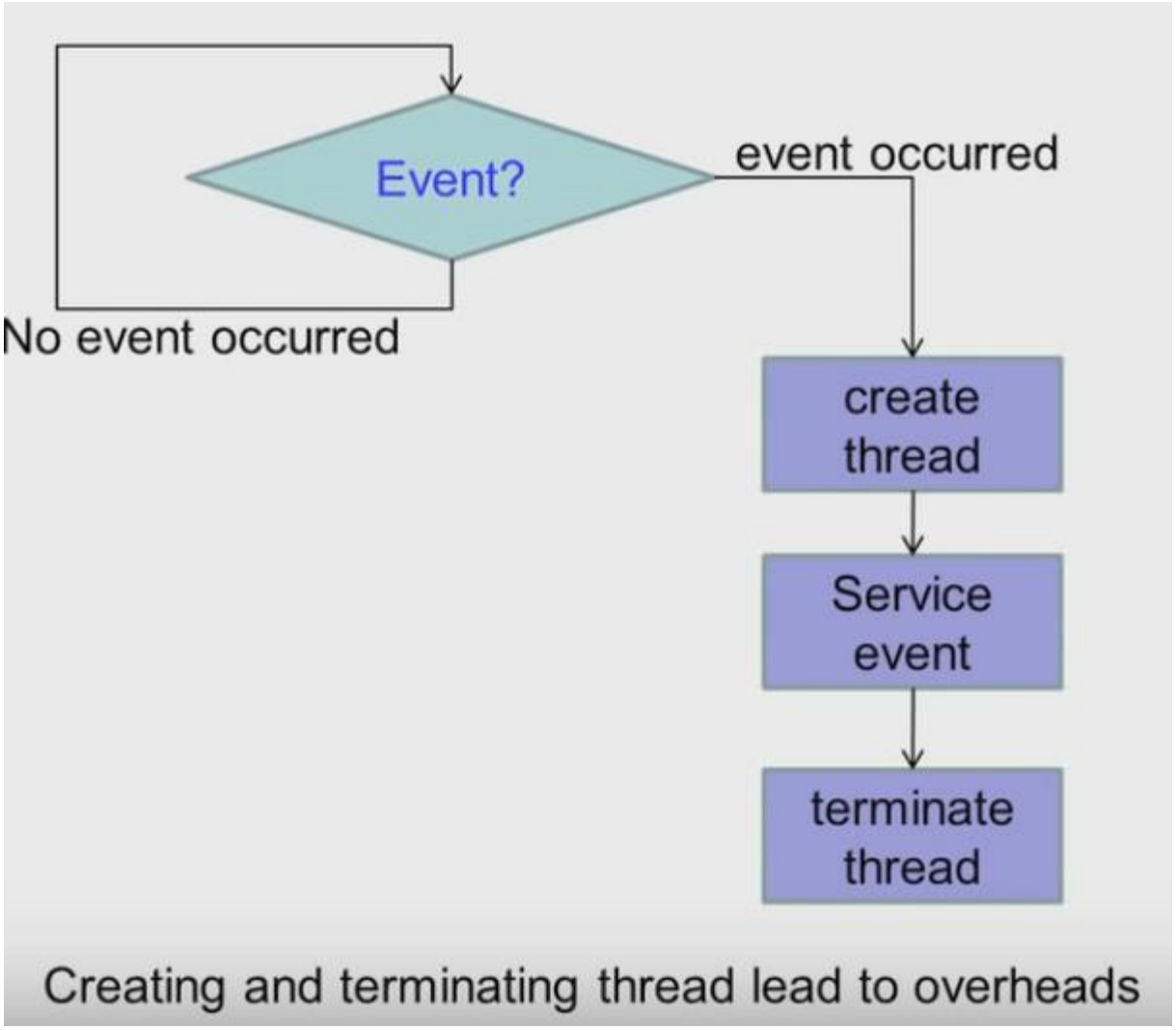
Design Issue

What happens when a thread invokes fork?

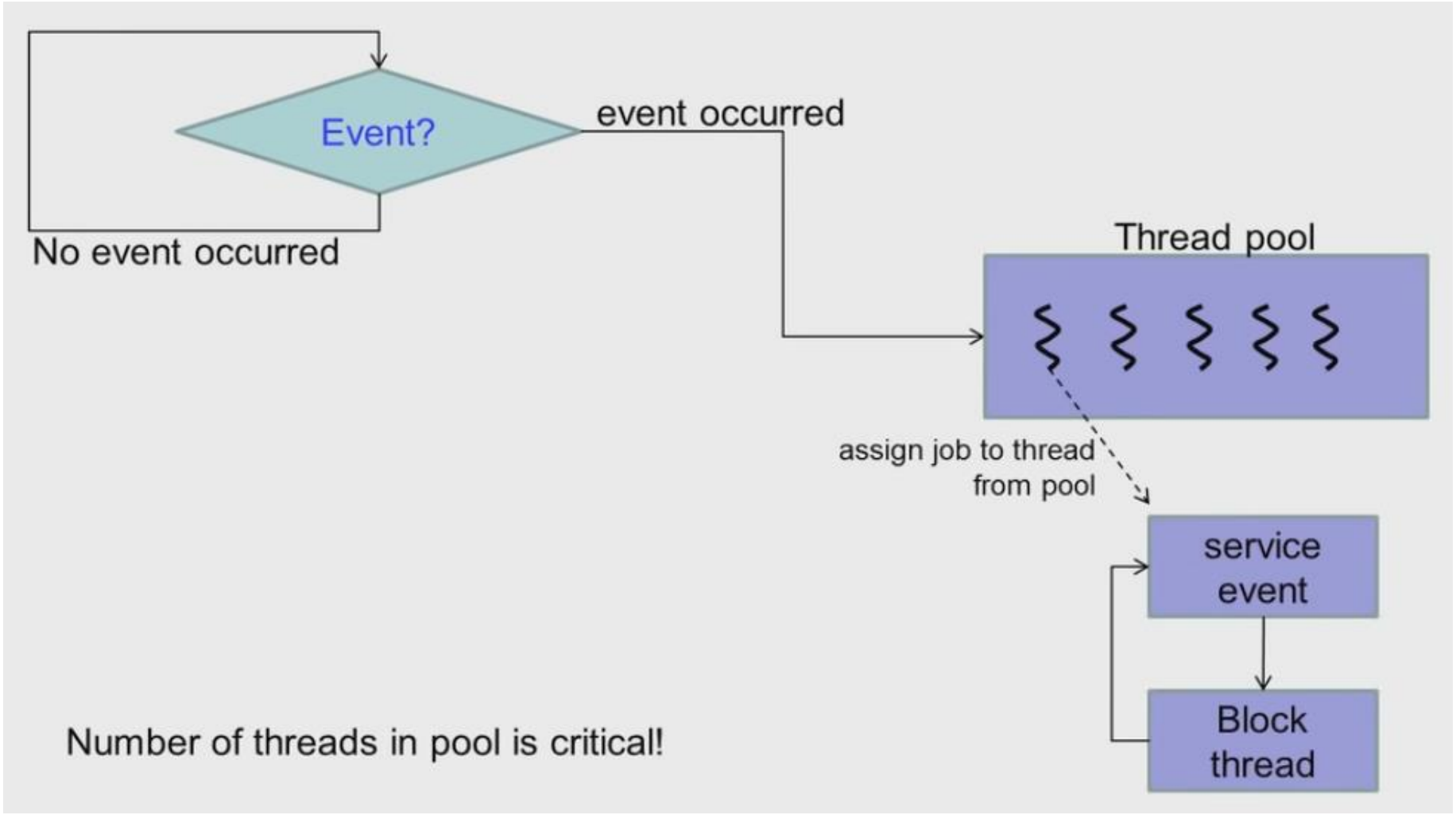
- Duplicate all threads?
  - Not easily done… other threads may be running or blocked in a system call or a critical section
- Duplicate only the caller thread?
  - More feasible.

Segmentation fault in a thread. Should only the thread terminate or the entire process?

Typical Usage



Thread Pool


Event?
event occurred
No event occurred
Thread pool
assign job to thread
from pool
service
event
Block
thread
Number of threads in pool is critical!

Pthread library

Thread identifier (TID) much like

Create a thread in a process

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

Pointer to a function, which starts execution in a different thread

Arguments to the function

Destroying a thread

```
void pthread_exit(void *retval);
```

Join : Wait for a specific thread to complete

```
int pthread_join(pthread_t thread, void **retval);
```

TID of the thread to wait for

Exit status of the thread

```c
#include <pthread.h>
#include <stdio.h>

unsigned long sum[4];

void *thread_fn(void *arg){
  long id = (long) arg;
  int start = id * 2500000;
  int i=0;

  while(i < 2500000){
     sum[id] += (i + start);
     i++;
  }
  return NULL;
}

int main(){
  pthread_t t1, t2, t3, t4;

  pthread_create(&t1, NULL, thread_fn, (void *)0);
  pthread_create(&t2, NULL, thread_fn, (void *)1);
  pthread_create(&t3, NULL, thread_fn, (void *)2);
  pthread_create(&t4, NULL, thread_fn, (void *)3);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  pthread_join(t4, NULL);
  printf("%lu\n", sum[0] + sum[1] + sum[2] + sum[3]);
  return 0;
}
```

```
$ gcc threads.c –lpthread
$ ./a.out
```