Processes and Threads

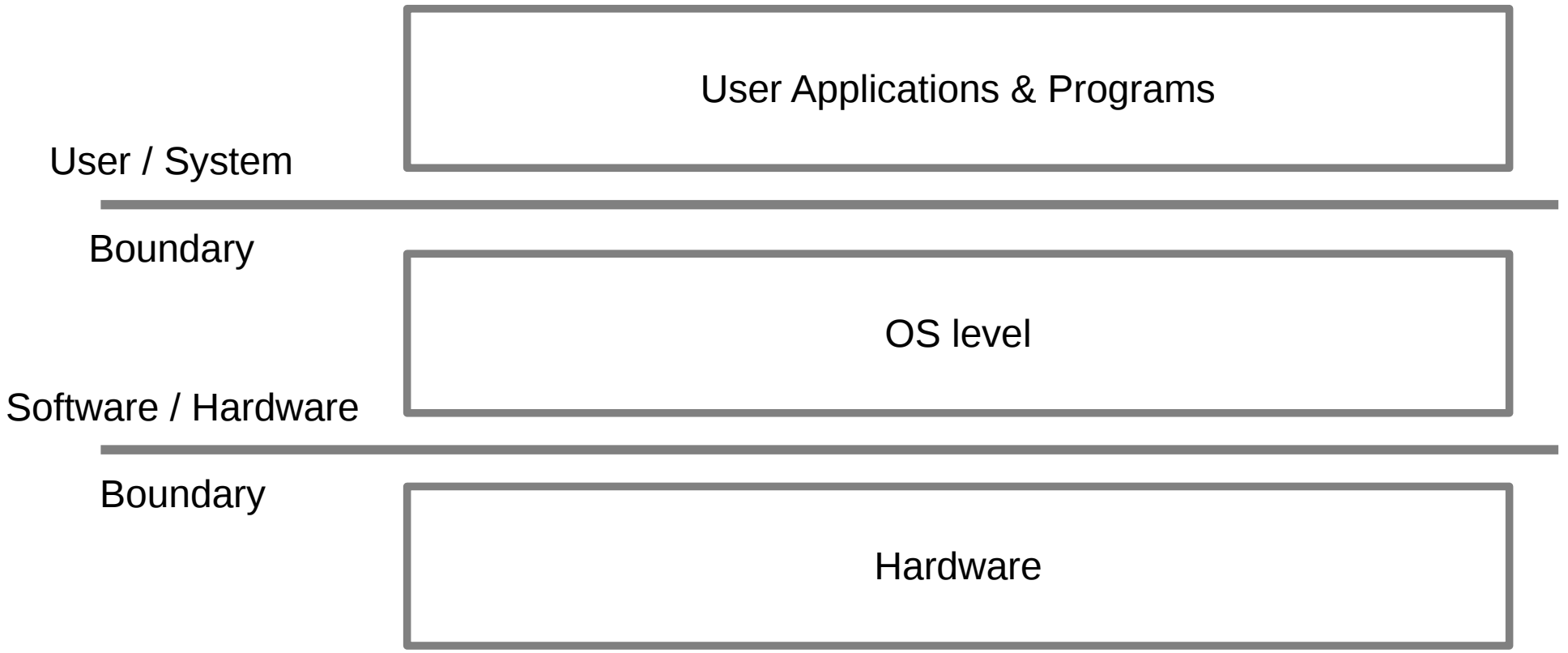
ECSE 420 - Tutorial 1

Dimitrios Stamoulis

TR 4110
September 22, 2014

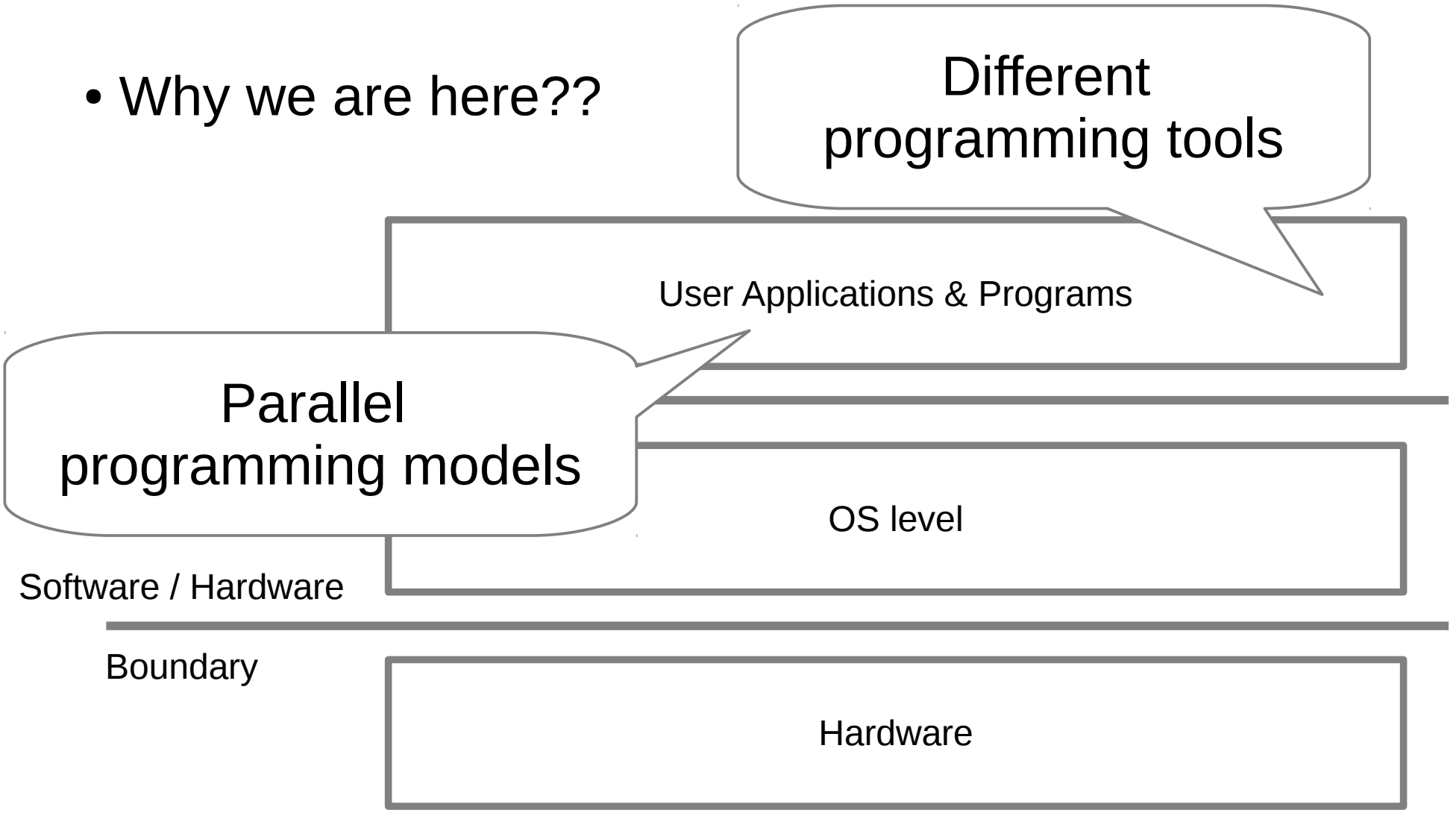
ECSE 420 Tutorials – Introduction

- Why we are here??



ECSE 420 Tutorials – Introduction

- Why we are here??



ECSE 420 – Tutorials (1/2)

- Hours and Location
  - Group 1 : Mondays, 04:00 PM - 5:30 PM (TR4110)
  - Group 2 : Wednesdays, 04:00 PM - 5:30 PM (TR4110)

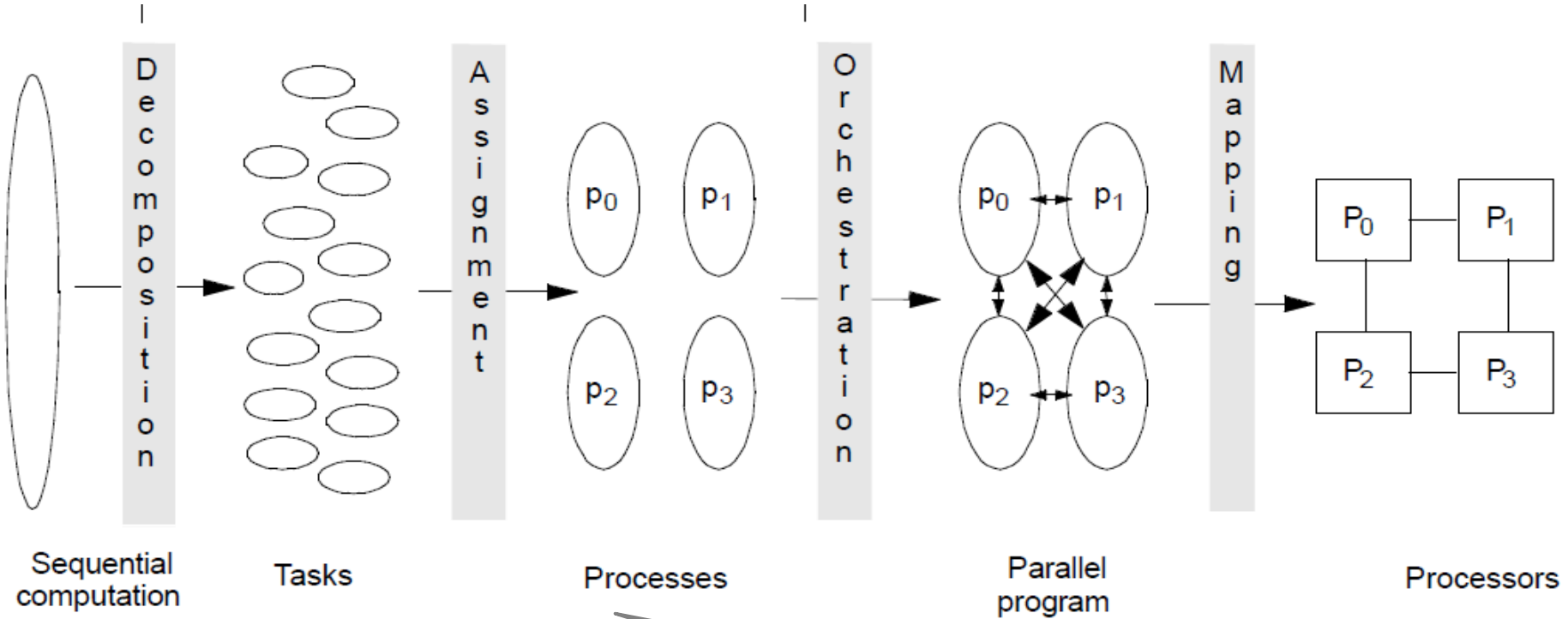
- Our goal to have a foretaste of:
  - Useful parallel programming tools (e.g OpenMP, MPI)
  - Midterm exercises, assignments etc

- So, please ask questions !!
  - Office Hours :
    Wednesdays, 10.00-11.30am, McConnell 544
  - Mail them @ :
    dimitrios.stamoulis@mail.mcgill.ca

ECSE 420 – Tutorials (2/2)

- Labs Schedule (tentative)
  - Tutorial 1 - 09/22 :
    Processes & Threads (Assignment 1)
  - Tutorial 2 - 09/29 :
    Processes & Threads (Lab 1)
  - Tutorial 3 - 10/06 :
    Shared Memory/ Msg Passing (Assignment 2)
  - Tutorial 4 - 10/13 :
    Shared Memory/ Msg Passing (Lab 2)
  - Tutorial 5 - 10/20 :
    Parallel/ Distributed Program. (Assignment 3)
  - Tutorial 6 – 10/27:
    Parallel/ Distributed Program. (Lab 3)

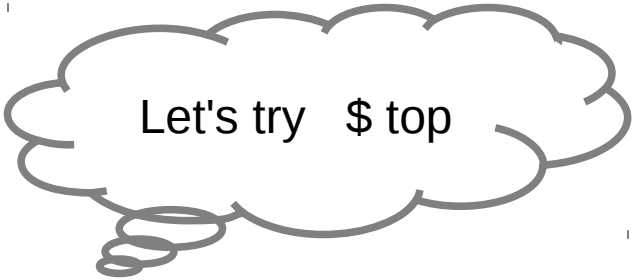
Tutorial 1 - Introduction



What's a process??

Process

- An 'alive' program
  - currently executed
  - inside an allocated memory portion (RAM)

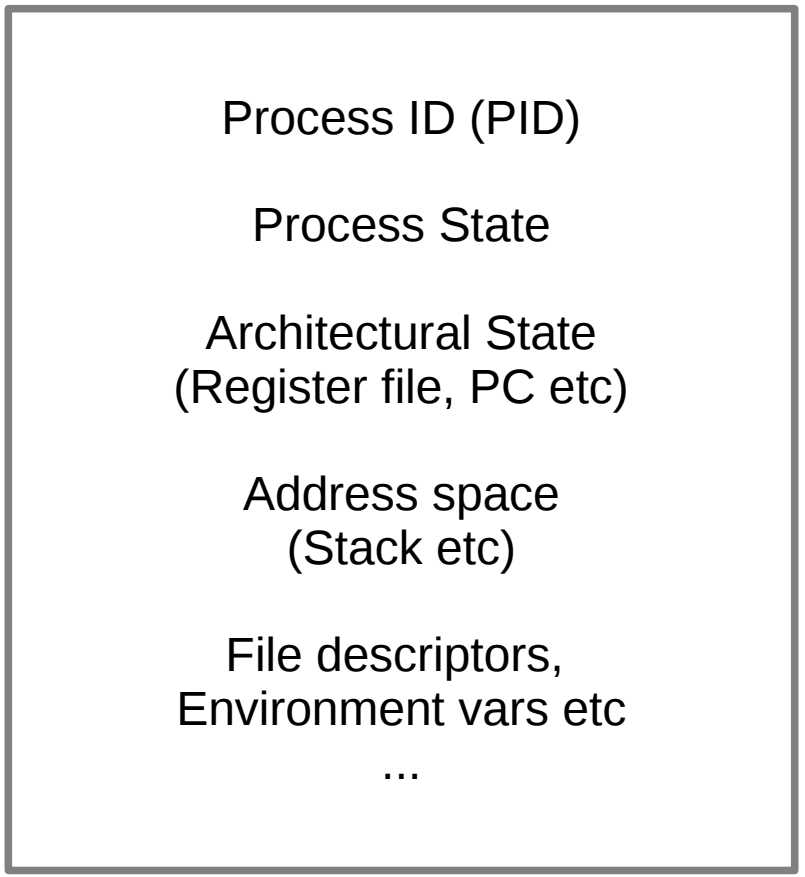


- Process = program (code) + state
  Many processes can execute the same code, but they have different execution states !!

- Unique Process ID (PID)

Process' State

- Process Control Block (PCB)



How to create a process : fork()

PID=832 ~ Parent

- **Program**

Same program

PID=864 ~ Child

- **Program**

- **State**

Process ID (PID)

Process State

Architectural State
(Register file, PC etc)

Address space
(Stack etc)

File descriptors,
Environment vars etc
...

- **State**

Process ID (PID) *NEW*

Process State *NEW*

Architectural State *NEW*
(Register file, PC etc)

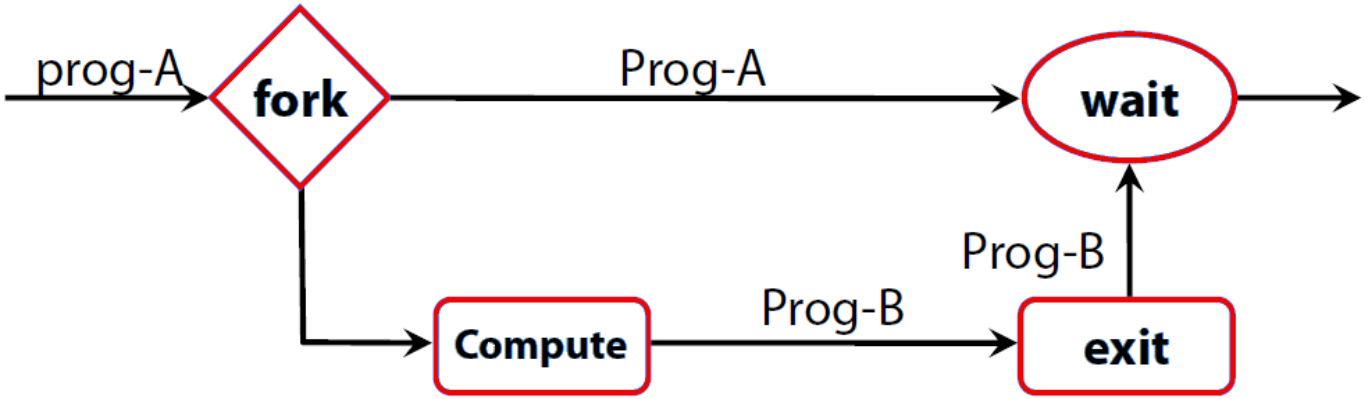
Address space *exact COPY*
(Stack etc)

File descriptors, *Inherited by child*
Environment vars etc
...

How to create a process : fork()

PID=832

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```



p = ? mypid = ?

How to create a process : fork()

PID=832

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = ? mypid = ?

How to create a process : fork()

PID=832

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
    perror("fork");
    exit(1);
} else if (p == 0) {
    mypid = getpid();
    child();
} else {
    father();
}
```

WHY??

p = -1 mypid = ?

How to create a process : fork()

PID=832

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = ? mypid = ?

How to create a process : fork()

PID=832

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = ? mypid = ?

How to create a process : fork()

PID=832

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = 864 mypid = ?

PID=864

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = 0 mypid = ?

How to create a process : fork()

PID=832

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
    perror("fork");
    exit(1);
} else if (p == 0) {
    mypid = getpid();
    child();
} else {
    father();
}
```

p = 864 mypid = ?

PID=864

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
    perror("fork");
    exit(1);
} else if (p == 0) {
    mypid = getpid();
    child();
} else {
    father();
}
```

p = 0 mypid = ?

How to create a process : fork()

PID=832

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = 864 mypid = ?

PID=864

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = 0 mypid = ?

How to create a process : fork()

PID=832

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = 864 mypid = ?

PID=864

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = 0 mypid = ?

How to create a process : fork()

PID=832

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = 864 mypid = ?

PID=864

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = 0 mypid = 864

How to create a process : fork()

PID=832

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = 864 mypid = ?

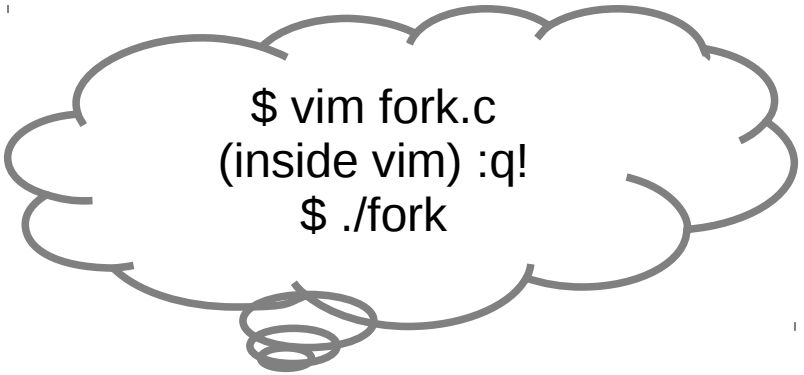
PID=864

```
pid_t p, mypid;
...
p = fork();
if (p < 0) {
     perror("fork");
     exit(1);
} else if (p == 0) {
     mypid = getpid();
     child();
} else {
     father();
}
```

p = 0 mypid = 864

Example - fork()

```
int main()
{
  pid_t fork(void);
  pid_t pid, child_pid, parent_pid;

  switch (pid = fork())
  {
    case -1:
        perror("The fork failed!");
        break;

    case 0:
        child_pid = getpid();
        printf("\nHello I am a new Child w/ PID %d\n",child_pid);
        break;

    default:
        parent_pid = getpid();
        printf("\nHello I am parent w/ PID %d and\nI created the process %d\n",parent_pid, pid);
    }

  return 0;
}
```



How to create a process : fork()

PID=832 ~ Parent

- **Program**

Same program

PID=864 ~ Child

- **Program**

- **State**

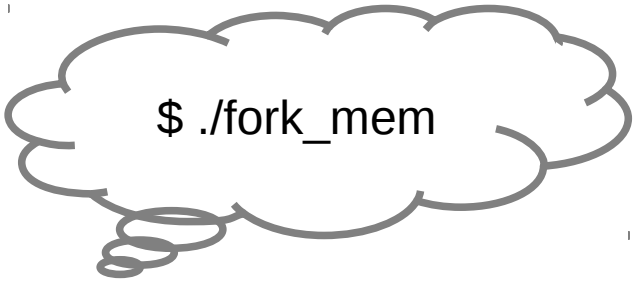
Process ID (PID)
Process State
Architectural State (Register file, PC etc)
Address space (Stack etc)
File descriptors, Environment vars etc ...

- **State**

Process ID (PID) *NEW*
Process State *NEW*
Architectural State *NEW* (Register file, PC etc)
Address space *exact COPY* (Stack etc)
File descriptors, *Inherited by child* Environment vars etc ...

Example - fork_mem()

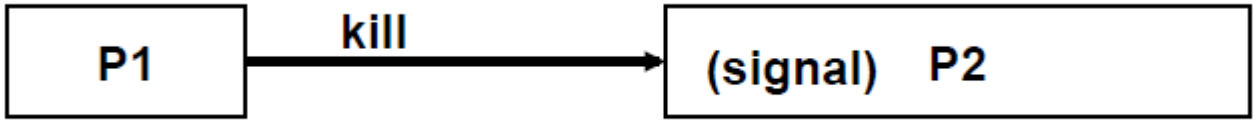
```
int main()
{

  pid_t pid, child_pid;
  int a[10]= {0,10,20,30,40,50,60,70,80,90};
  int i;

  switch (pid = fork())
  {
    case -1:
        perror("The fork failed!");
        break;

    case 0:
        child_pid = getpid();
        printf("Hello I am a new Child my pid is %d\n",child_pid);
        a[4]= 50;
        break;

    default:
        printf("Hello I am parent I just created a new process %d\n",pid);
        a[4]= 30;
    }

    for ( i = 0; i < 10; i++)
        printf("a[%d]=%d\n", i , a[i]);

  return 0;
}
```



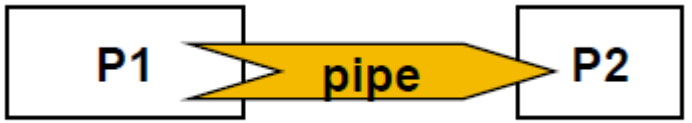
Inter-Process Communication (IPC)

IPC Mechanisms

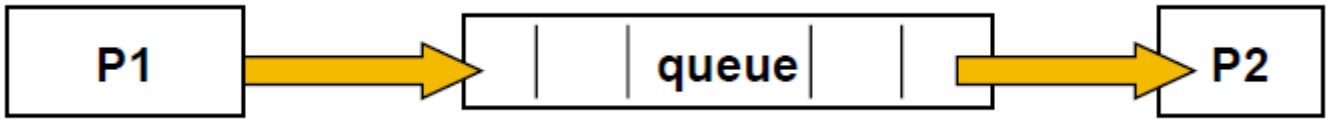
- Signals



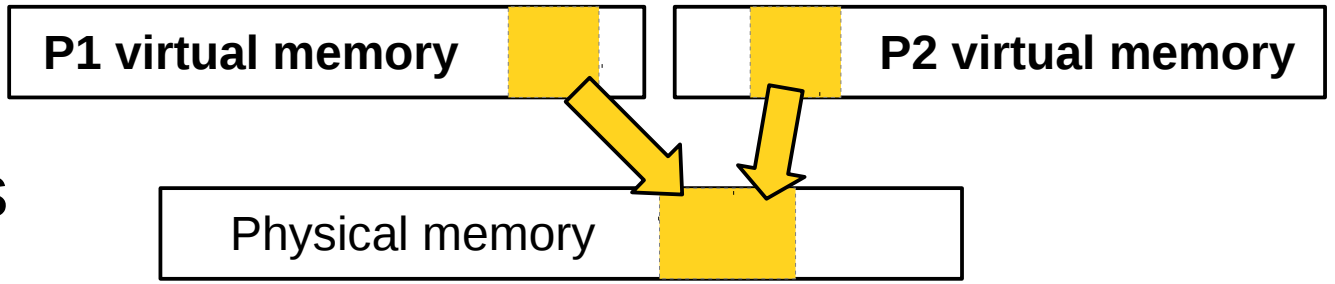
- Pipes



- POSIX messages



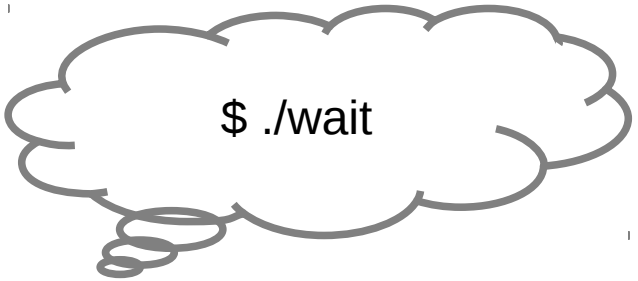
- Shared Memory Segments



Example - wait()

- Execute $ ./wait

```
Parent process started at Mon Sep 30 02:47:07 2013
Parent waiting for child at Mon Sep 30 02:47:07 2013
Child w/ PID 8580 started at Mon Sep 30 02:47:07 2013
Parent waiting for child at Mon Sep 30 02:47:08 2013
...
Child ended normally
```



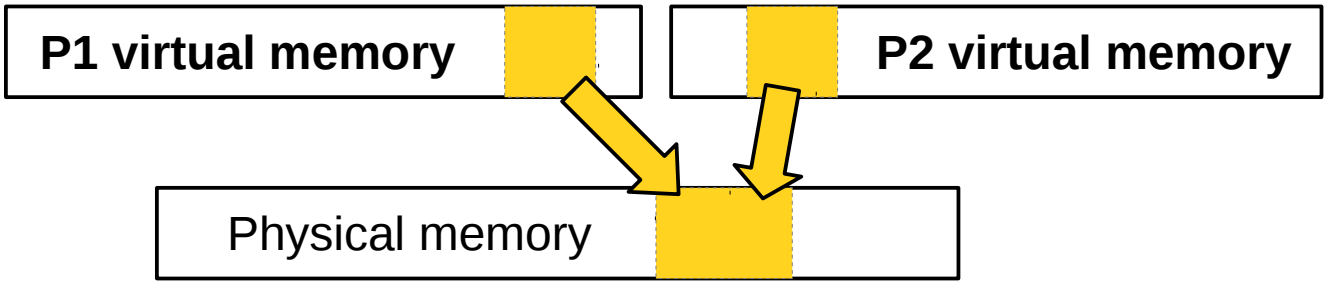
- What if i stop the child?
  - kill -STOP 8580
- What if i kill the child?
  - kill -9 8580
- What if i kill the parent?

I can use a signal as an execution checkpoint..!!

We can extend this idea to interprocess messages :

int MPI_Bcast()

Shared memory → Critical Sections




```
15.       while (!done) do                    /*outer loop sweeps*/
16.          mydiff = diff = 0;               /*set global diff to 0 (okay for all to do it)*/
16a.         BARRIER(bar1, nprocs);           /*ensure all reach here before anyone modifies diff*/
17.          for i ← mymin to mymax do        /*for each of my rows*/
18.            for j ← 1 to n do              /*for all nonborder elements in that row*/
19.                 temp = A[i,j];
20.                 A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                                 A[i,j+1] + A[i+1,j]);
22.                 mydiff += abs(A[i,j] - temp);
23.            endfor
24.          endfor
25a.         LOCK(diff_lock);
               diff += mydiff;
25c.         UNLOCK(diff_lock);
25d.         BARRIER(bar1, nprocs);
25e.         if (diff/(n*n) < TOL) then done

25f.         BARRIER(bar1, nprocs);
```

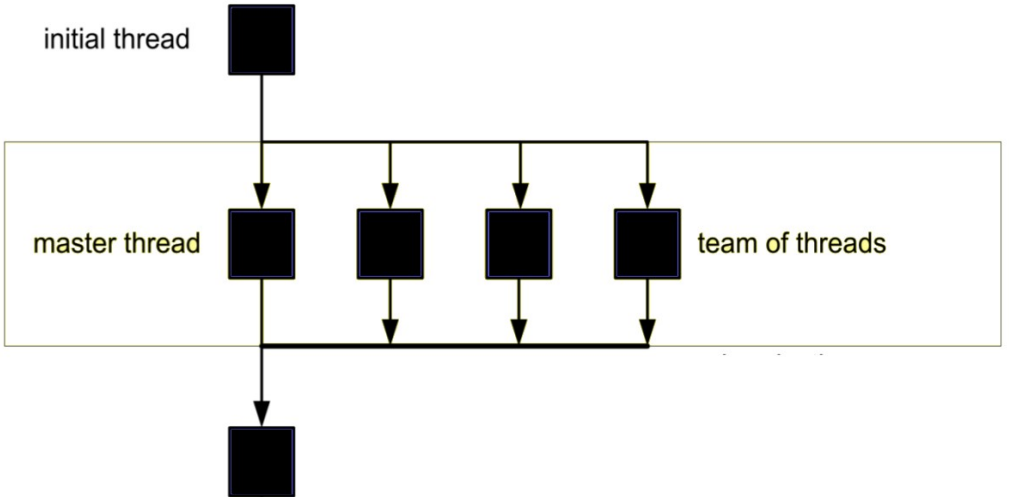
We will see same techniques :

- OpenMP :
  #pragma omp critical

Threads (1/2)

- A thread exists within a process :
  A finer-grained unit of execution than processes

- $ ./myprogram → Linux creates a new process → Linux creates a single, master thread.



- The master thread can create additional threads.

Threads (2/2)

- Threads within a process use process' resources :
  - Share process' resources
    Is it always good ? → NO !!
    WHY? → explicit synchronization by the programmer
  - Duplicate only architectural state
    WHY? → To be independently schedulable

Threads vs Processes

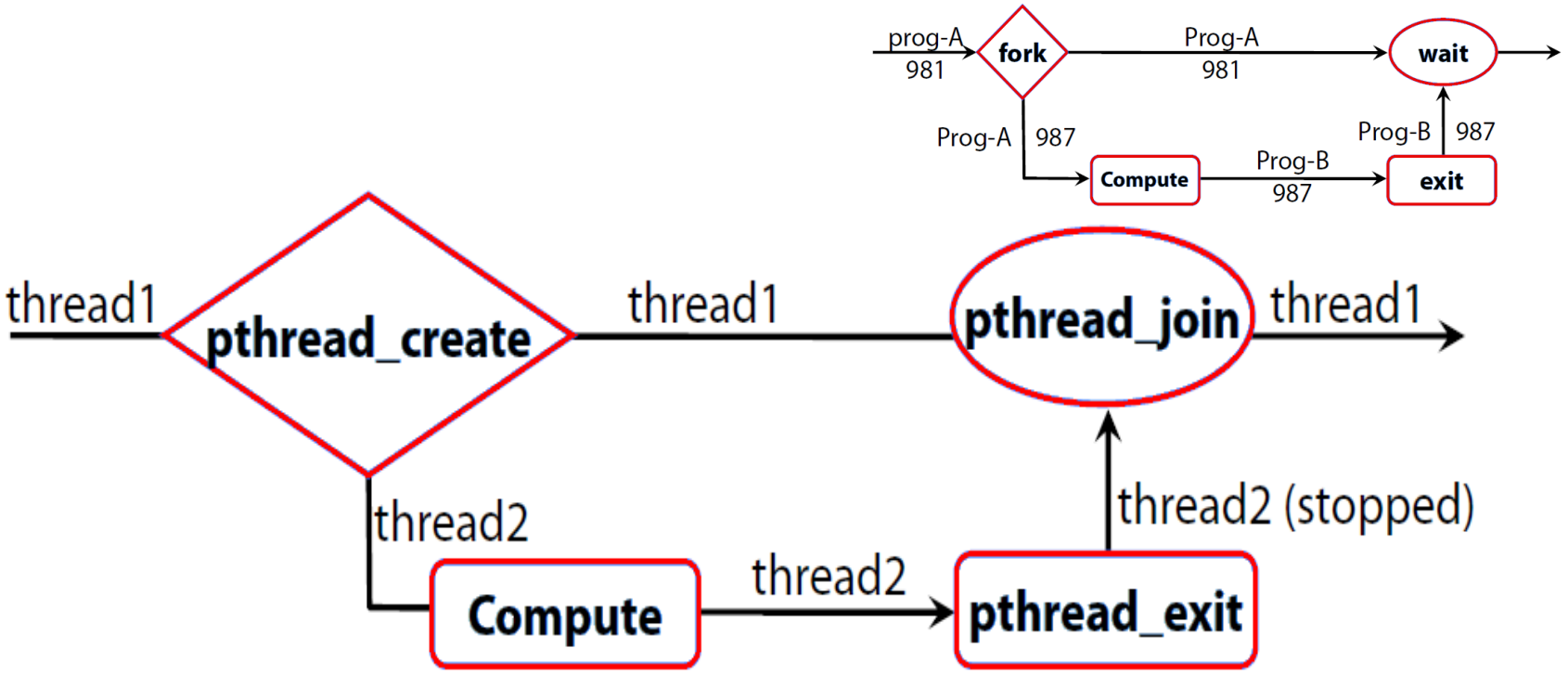
- Thread creation is “lightweight” - WHY?
  - A thread uses less resources than a process.

- Less communication overhead
- Less scheduling overhead

- More efficient programming techniques / tools :
  - e.g : OpenMP

```
#pragma omp parallel num_threads(4)
{
    ....
}
```

Pthreads (POSIX threads) - pthread_create()

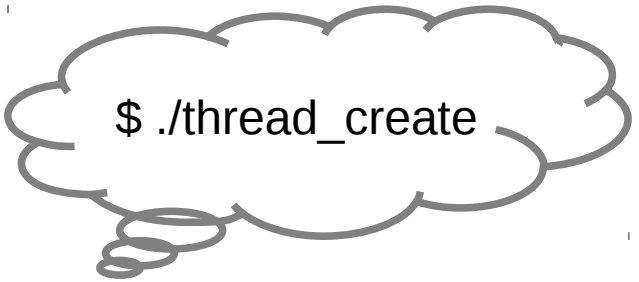


pthread_create(thread,attr,start_routine,arg) arguments:

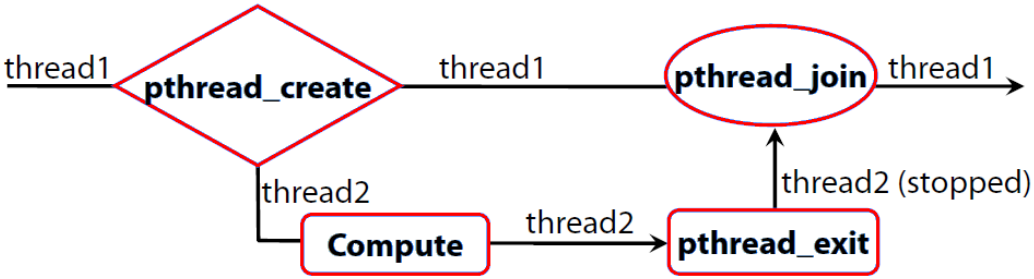
- thread: Identifier for the new thread.
- Attr: Attribute object (e.g Stack size, priority). NULL for the default values!!
- start_routine: the C routine that the thread will execute.
- arg: A single argument that may be passed to start_routine.

How to create a thread : pthread_create()

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```



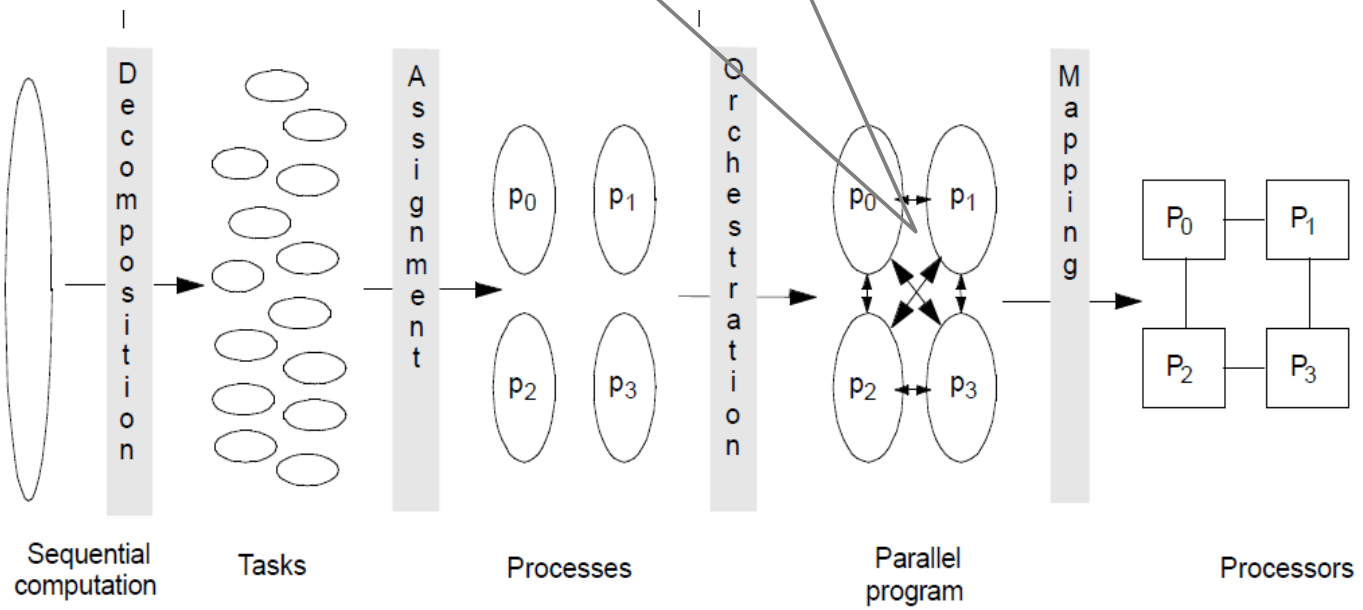
Pthreads (POSIX threads) - pthread_join()



pthread_join(threadid,status) :

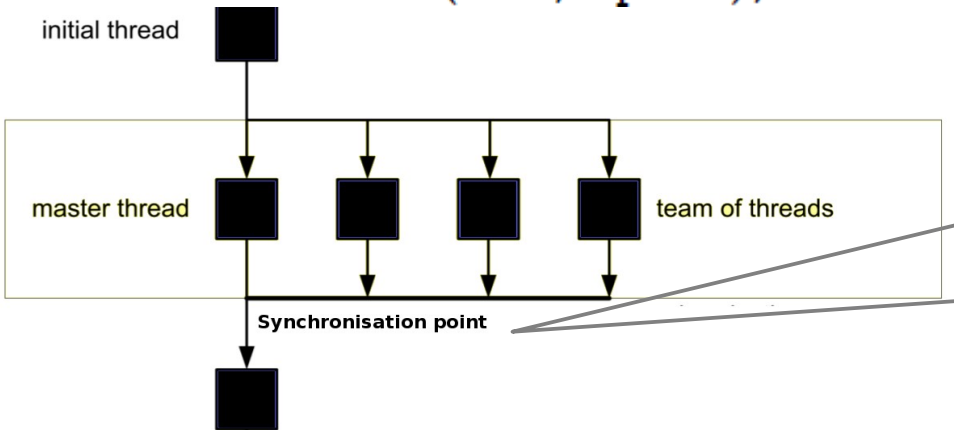
- Gives the **threadid** termination return status if it was specified in the **threadid** call to pthread_exit().
- pthread_join() blocks the calling thread until the specified **threadid** terminates.

WHY is that important? → **Synchronization Point !!**



pthread_join() - Synchronization

```
15.      while (!done) do                          /*outer loop sweeps*/
16.        mydiff = diff = 0;                      /*set global diff to 0 (okay for all to do it)*/
16a.       BARRIER(bar1, nprocs);                  /*ensure all reach here before anyone modifies diff*/
17.        for i ← mymin to mymax do               /*for each of my rows*/
18.          for j ← 1 to n do                     /*for all nonborder elements in that row*/
19.               temp = A[i,j];
20.               A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                         A[i,j+1] + A[i+1,j]);
22.               mydiff += abs(A[i,j] - temp);
23.          endfor
24.        endfor
25a.       LOCK(diff_lock);                              /*update global diff if necessary*/
25b.         diff += mydiff;
25c.       UNLOCK(diff_lock);
           BARRIER(bar1, nprocs);                  /*ensure all reach here before checking if done*/
25e.       if (diff/(n*n) < TOL) then done = 1;    /*check convergence; all get
                                                     same answer*/
25f.       BARRIER(bar1, nprocs);
```



We will see same techniques :

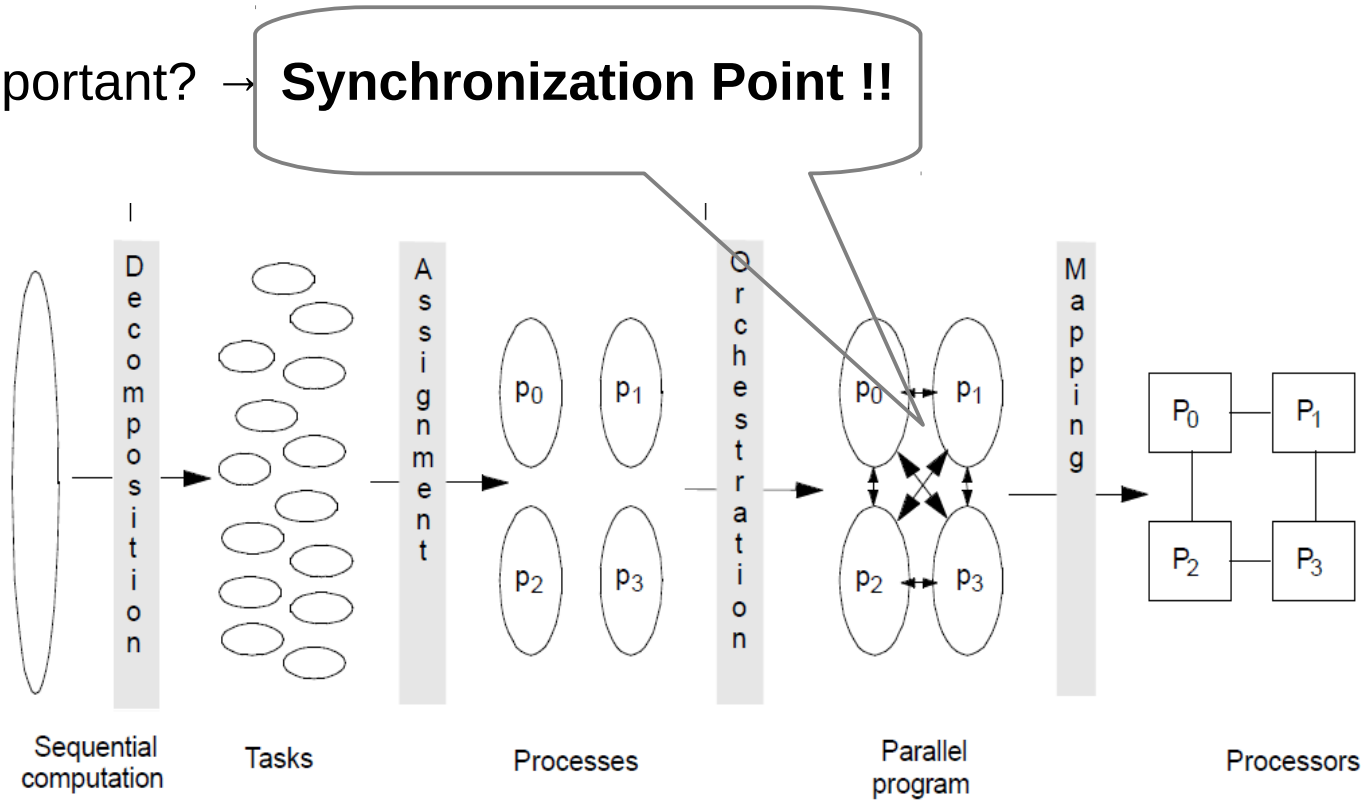
- OpenMP :
  #pragma omp barrier
- MPI :
  int MPI_Barrier()

Joining threads : pthread_join()


```
int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   int rc;
   long t;
   void *status;

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], NULL, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
         }
      }

   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
         }
      printf("Main: completed join with thread %ld having a status of %ld\n",t,(long)status);
      }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

Synchronization (1/2)

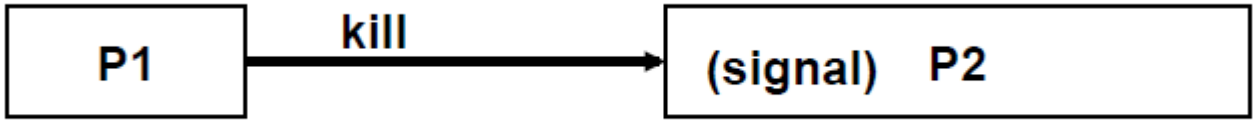
WHY is that important? → **Synchronization Point !!**



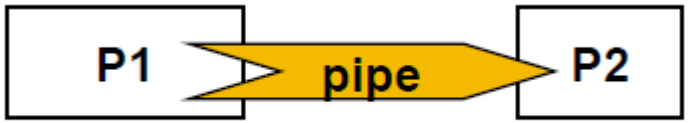
Synchronization (2/2)

Inter-Process Communication (IPC) Mechanisms

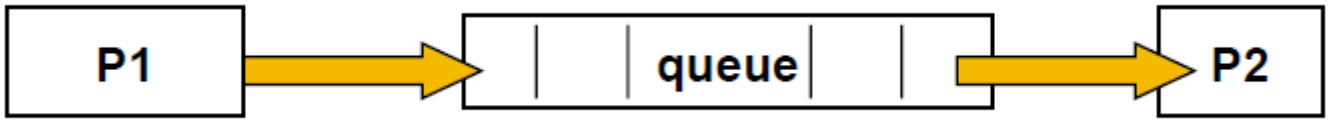
- Signals



- Pipes



- POSIX messages



- Shared Memory Segments