R

بسم الله الرحمن الرحیم

سیستم عامل

جلسه سوم – ادامه‌ی پردازه، ریسمان

Original slides by: Jonathan Walpole & Hadi Foroughmand Arabi – Some slides/images by Silberschatz, Galvin and Gagne

یادآوری!

- دوشنبه آزمونک اول است!

آنچه گذشت

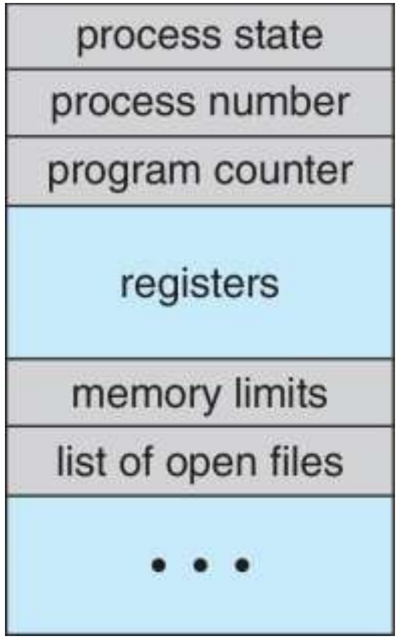
جلسه‌ی قبل، پردازه

The Process Concept (Vs. Program)

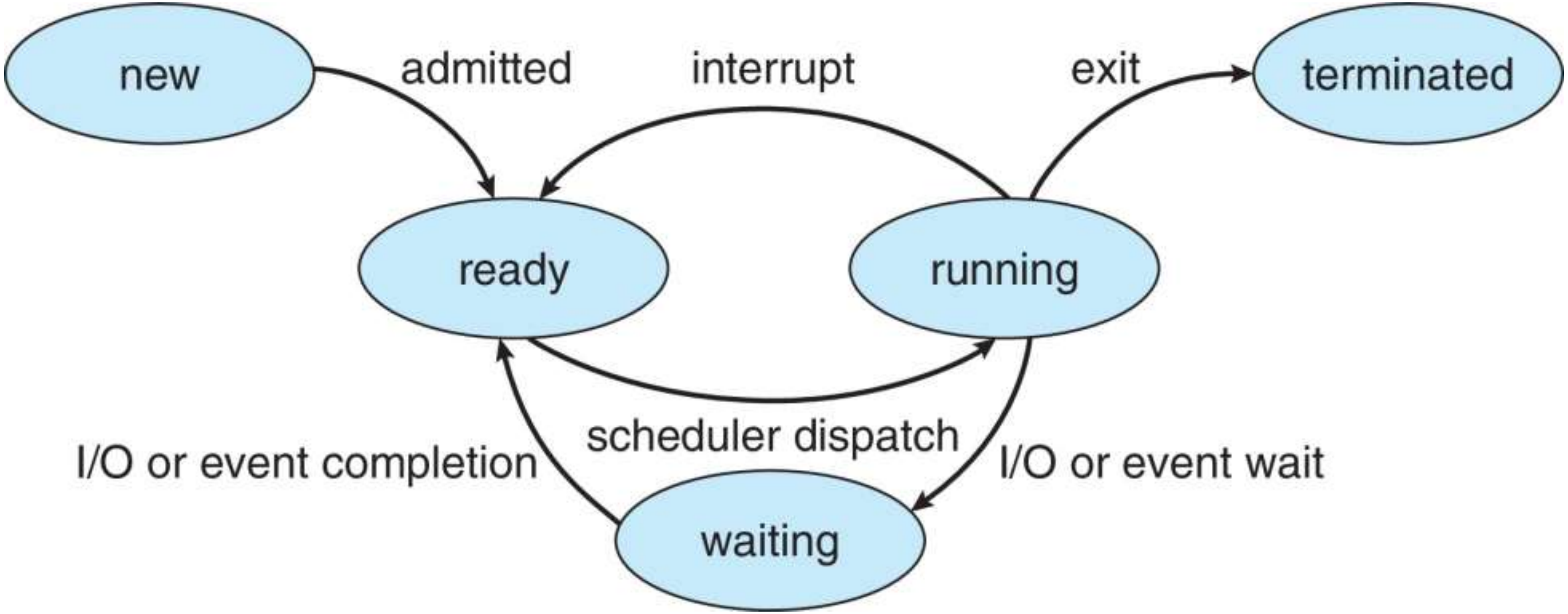
- Process – a program in execution
- Program
  - description of how to perform an activity
  - instructions and static data values
- Process
  - a snapshot of a program in execution
  - memory (program instructions, static and dynamic data values)
  - CPU state (registers, PC, SP, etc)
  - operating system state (open files, accounting statistics etc)

Process Control Block (PCB)

- **Process state** – running, waiting, etc.
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information-** priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files



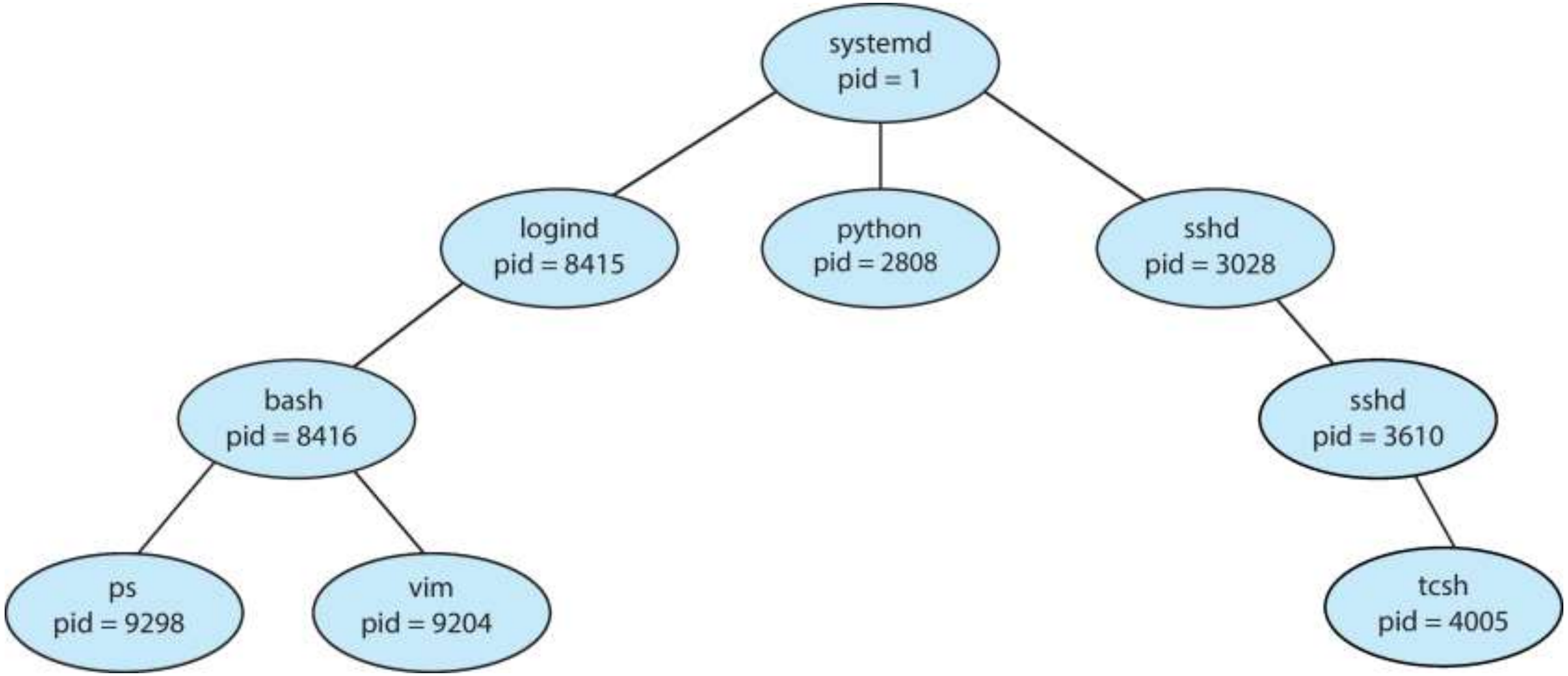
Process State in More Detail



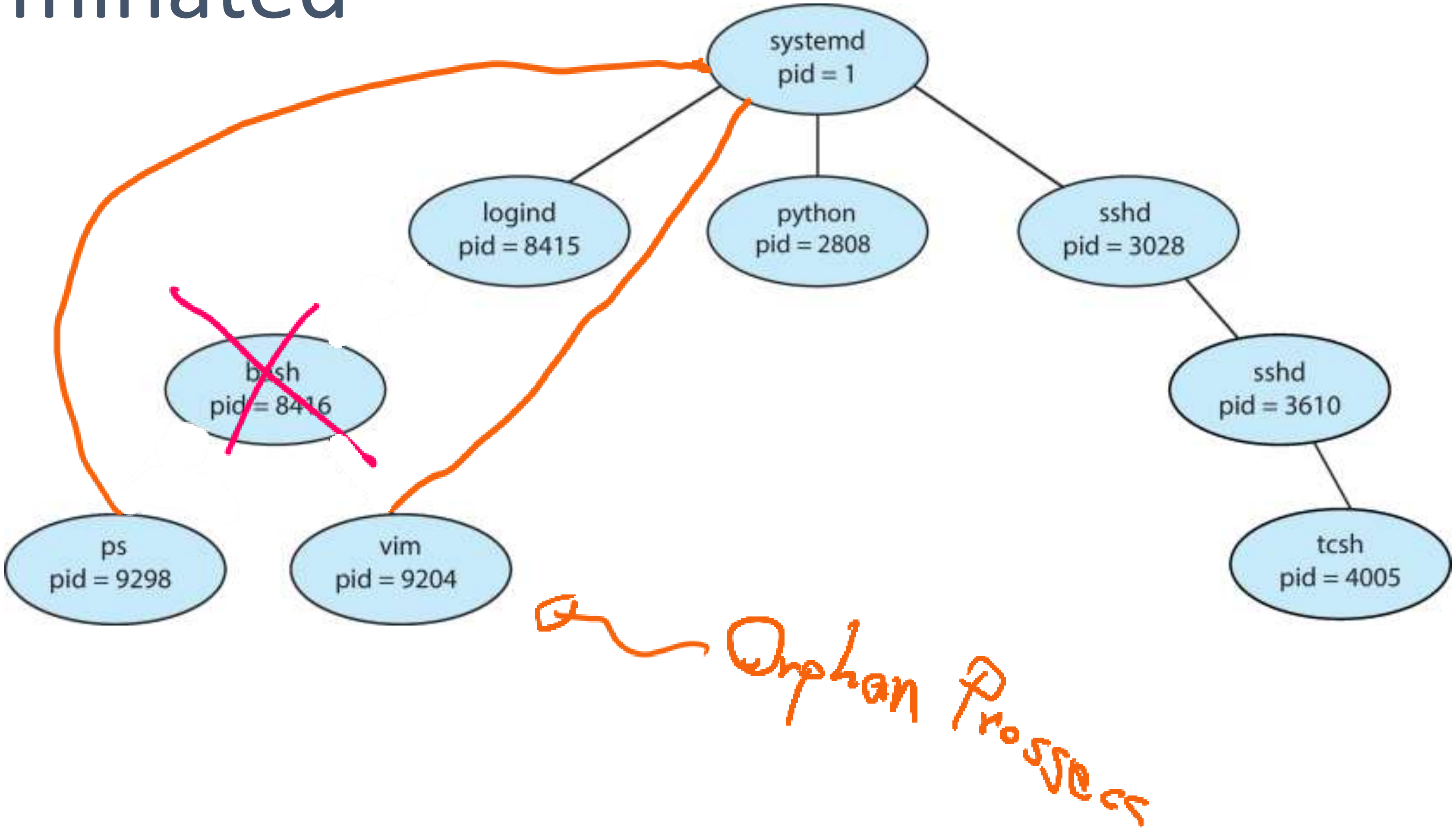
Question

- Why there are new and terminated states ?
- New State:
  - *Prevents the system from scheduling an incomplete or uninitialized process.*
- Terminated State:
  - *Ensures that resources used by the process are properly cleaned up and that exit statuses are correctly handled.*
- Zombie process:
  - *The parent wait() to collect exit status*
  - *Child process in terminated states (Zombie)*

A Tree of Processes in Linux



What Happen if a parent process terminated



Process Creation in UNIX

- All processes have a unique process id
  - ***getpid()****, ****getppid()**** system calls allow processes to get their information*
- Process creation
  - ***fork()**** system call creates a copy of a process and returns in both processes (parent and child), but with a different return value*
  - ***exec()**** replaces an address space with a new program*
- Process termination, signaling
  - ***signal()****, ****kill()**** system calls allow a process to be terminated or have specific signals sent to it*

Process Creation Example

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
exec("/bin/ls");
  }
else {
  // parent
  wait();
  }
…
```

Process Creation Example

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
  exec(“/bin/ls”);
  }
else {
  // parent
  wait();
  }
…
```

csh (pid = 24)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
  exec(“/bin/ls”);
  }
else {
  // parent
  wait();
  }
…
```

Process Creation Example

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
  exec(“/bin/ls”);
  }
else {
  // parent
  wait();
  }
…
```

csh (pid = 24)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
  exec(“/bin/ls”);
  }
else {
  // parent
  wait();
  }
…
```

Process Creation Example

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec(“/bin/ls”);
   }
else {
   // parent
   wait();
   }
…
```

csh (pid = 24)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec(“/bin/ls”);
   }
else {
   // parent
   wait();
   }
…
```

Process Creation Example

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec(“/bin/ls”);
   }
else {
   // parent
   wait();
   }
…
```

ls (pid = 24)

```
//ls program

main(){

   //look up dir

   …

}
```

Process Creation (fork)

- Fork creates a new process by *copying* the calling process
- The new process has its own
  - *Memory address space (copied from parent)*
    - Instructions (same program as parent!)
    - Data
    - Stack
  - *Register set (copied from parent)*
  - *Process table entry in the OS*

Fork Challenge!

What is the output of the program?

```
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Parent process started with PID: %d\n", getpid());

    fork();
    fork();
    printf("Process with PID: %d, Parent PID: %d\n", getpid(), getppid());

    return 0;
}
```

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

2

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

2

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

2

```
Parent process started with PID: 2
```

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

2

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
```

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

2

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
```

2

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

4

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

```
Parent process started with PID: 2
```

2

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID:
%d\n", getpid(), getppid());
return 0;
```

4

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
```

2

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID:
%d\n", getpid(), getppid());
return 0;
```

4

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
```

2

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID:
%d\n", getpid(), getppid());
return 0;
```

3

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

4

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID:
%d\n", getpid(), getppid());
return 0;
```

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
```

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID:
%d\n", getpid(), getppid());
return 0;
```

2

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

4

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
```

2

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID:
%d\n", getpid(), getppid());
return 0;
```

3

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

4

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(),
getppid());
return 0;
```

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
```

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

2

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(),
getppid());
return 0;
```

4

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
```

1

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(),
getppid());
return 0;
```

3

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

4

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(),
getppid());
return 0;
```

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
```

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(),
getppid());
return 0;
```

1

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
```

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
```

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
```

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(), getppid());
return 0;
```

6

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
```

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

6

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

6! or 5?

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
```

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(), getppid());
return 0;
```

6

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
Process with PID: 3, Parent PID: 1
```

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

6

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID:
%d\n", getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
Process with PID: 3, Parent PID: 1
```

← !!

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

6

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID:
%d\n", getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
Process with PID: 3, Parent PID: 1
```

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(), getppid());
return 0;
```

6

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
Process with PID: 3, Parent PID: 1
Process with PID 6, Parent PID: 3
```

```c
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

6

```c
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID:
%d\n", getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
Process with PID: 3, Parent PID: 1
Process with PID 6, Parent PID: 3
```

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(),
getppid());
return 0;
```

6

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID:
%d\n", getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
Process with PID: 3, Parent PID: 1
Process with PID 6, Parent PID: 3
```

```
printf("Parent process started with PID: %d\n", getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n", getpid(),
getppid());
return 0;
```

6

```
printf("Parent process started with PID: %d\n",
getpid());
fork();
fork();
printf("Process with PID: %d, Parent PID: %d\n",
getpid(), getppid());
return 0;
```

3

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
Process with PID: 3, Parent PID: 1
Process with PID 6, Parent PID: 3
```

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
Process with PID: 3, Parent PID: 1
Process with PID 6, Parent PID: 3
```

Is it the only feasible output?

```
Parent process started with PID: 2
Process with PID: 2, Parent PID: 10
Process with PID: 4, Parent PID: 2
Process with PID: 3, Parent PID: 1
Process with PID 6, Parent PID: 3
```

Combinatorial Problem

- How many ways that the code can run?

Questions?

جلسه‌ی جدید

ادامه‌ی پردازه، ریسمان

تمام شدن یک پردازه

How Do Processes Terminate?

Conditions that terminate processes:

- *Normal exit (voluntary)*
- *Error exit (voluntary)*
- *Fatal error (involuntary)*
- *Killed by another process (involuntary)*

Killing a process

- Sending kill signal to kernel
- Killing a process does not kill its descendants

wait()

- Waits until:
  - *A child is killed/terminated, or*
  - *A signal is received from OS (Interrupt the wait)*

Some important signals in Linux

- **SIGINT** (Interrupt): Sent when the user interrupts a process (usually with Ctrl+C).
- **SIGKILL** (Kill): Immediately terminates the process. Cannot be ignored or handled by the process.
- **SIGTERM** (Terminate): Requests the process to gracefully terminate. Can be caught to allow cleanup before exiting.
- **SIGSTOP** (Stop): Stops a process execution. Can be resumed later with **SIGCONT**.

پردازه‌ها و هم‌زمانی

- مثال:
  - *فرض کنید یک گراف داریم و می‌خواهیم به طور هم‌زمان الگوریتمی مثل DFS را از رئوس آن اجرا کنیم.*
- برنامه‌ی آن چه شکلی است؟

پردازه‌ها و هم‌زمانی - کد نمونه

```
Load graph and edges
let U be the set of start vertices for DFS
for (int v in U) {
   Int f = fork();
   If (fork() == 0) { // child process
      DFS(v);
      return 0;
   }
}
return 0;
```

پردازه‌ها و هم‌زمانی - کد نمونه

- میزان حافظه‌ی مصرفی با فرض اینکه حجم گراف ۱ گیگابایت باشد؟
- پردازه‌های فرزند، چطوری نتیجه‌ی اجرا را به پردازه‌ی والد بدهند؟

```
Load graph and edges
let U be the set of start vertices for DFS
for(int v in U):
     Int f = fork()
     If (fork() == 0) { // child process
         DFS(v)
         return 0;
     }
}
return 0;
```

ریسمان

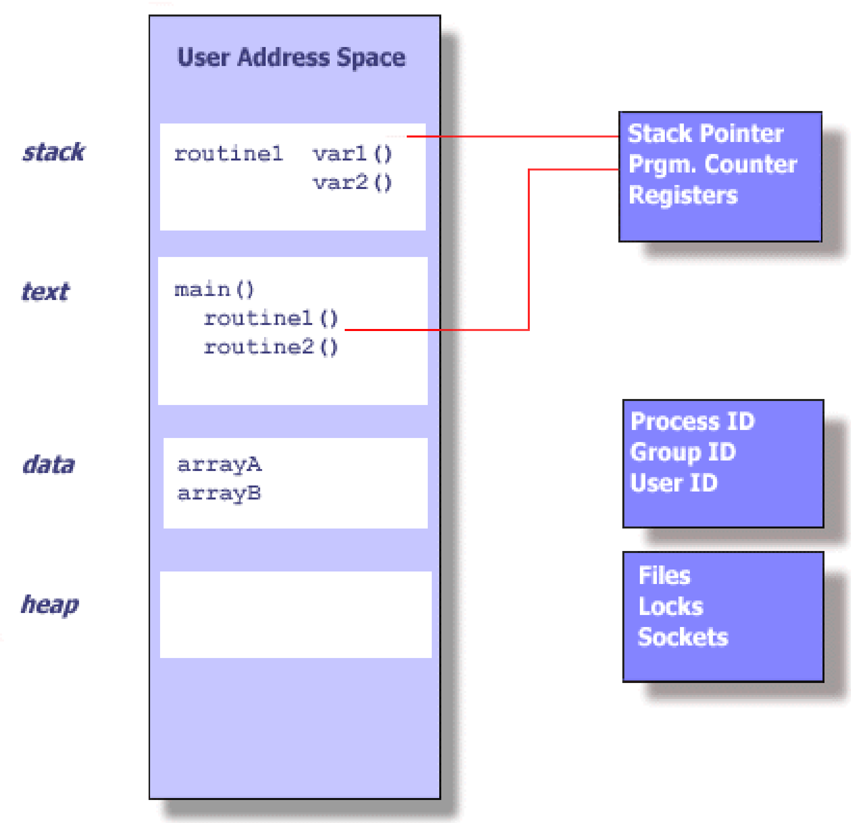
Threads

- Processes have the following components:
  - *an address space*
  - *a collection of operating system state*
  - *a CPU context … or thread of control*
- To use multiple CPUs on a multiprocessor system, a process would need several CPU contexts
  - *Thread fork creates new thread not memory space*
  - *Multiple threads of control could run in the same memory space on a single CPU system too!*

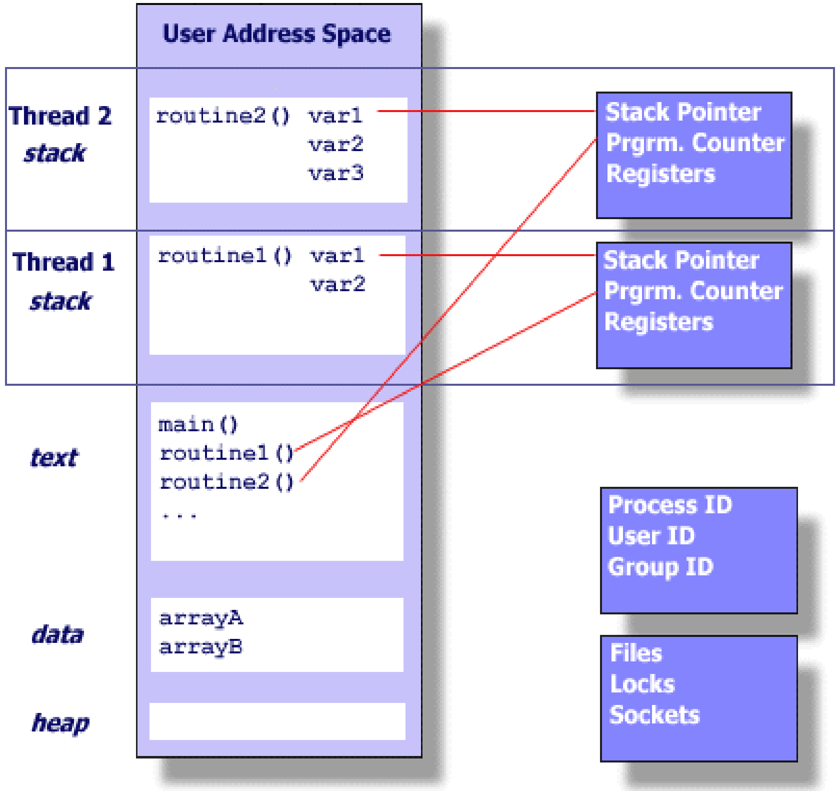
Threads

- Threads share a process address space with zero or more other threads
- Threads have their own CPU context
  - *PC, SP, register state*
  - *Stack*
- A traditional process could be viewed as a memory address space with a single thread

Single Thread in Address Space



Multiple Threads in Address Space



What Is a Thread?

- A thread executes a stream of instructions
  - *it is an abstraction for control-flow*
- Practically, it is a processor context and stack
  - *Allocated a CPU by a scheduler*
  - *Executes in a memory address space*

Private Per-Thread State

Things that define the state of a particular flow of control in an executing program

- *Stack (local variables)*
- *Stack pointer*
- *Registers*
- *Scheduling properties (i.e., priority)*

Shared State Among Threads

Things that relate to an instance of an executing program

- *User ID, group ID, process ID*
- *Address space:*
  - Text
  - Data (off-stack global variables)
  - Heap (dynamic data)
- *Open files, sockets, locks*

Concurrent Access to Shared State

Changes made to shared state by one thread will be visible to the others!

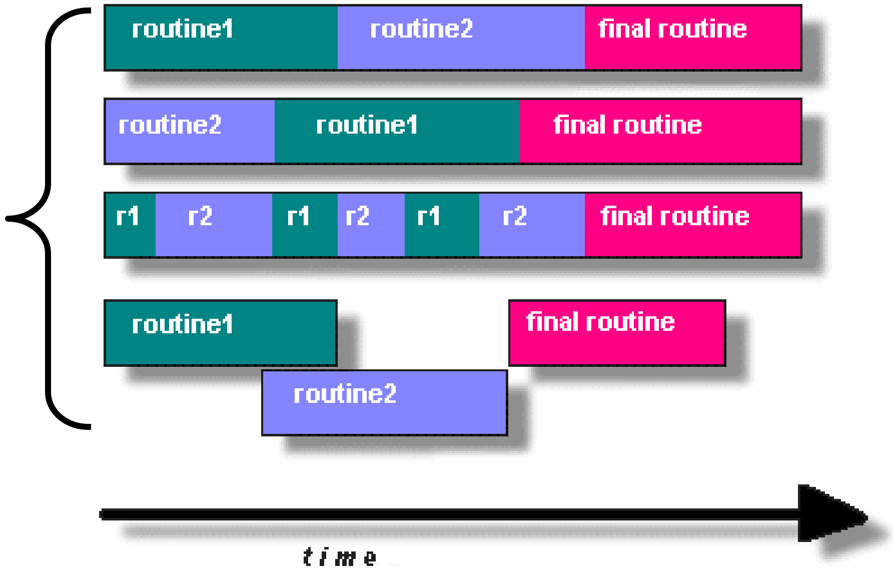
- Reading and writing memory locations requires synchronization!
- This is a major topic for later …

Programming With Threads

Split program into routines to execute in parallel

- *True or pseudo (interleaved) parallelism*

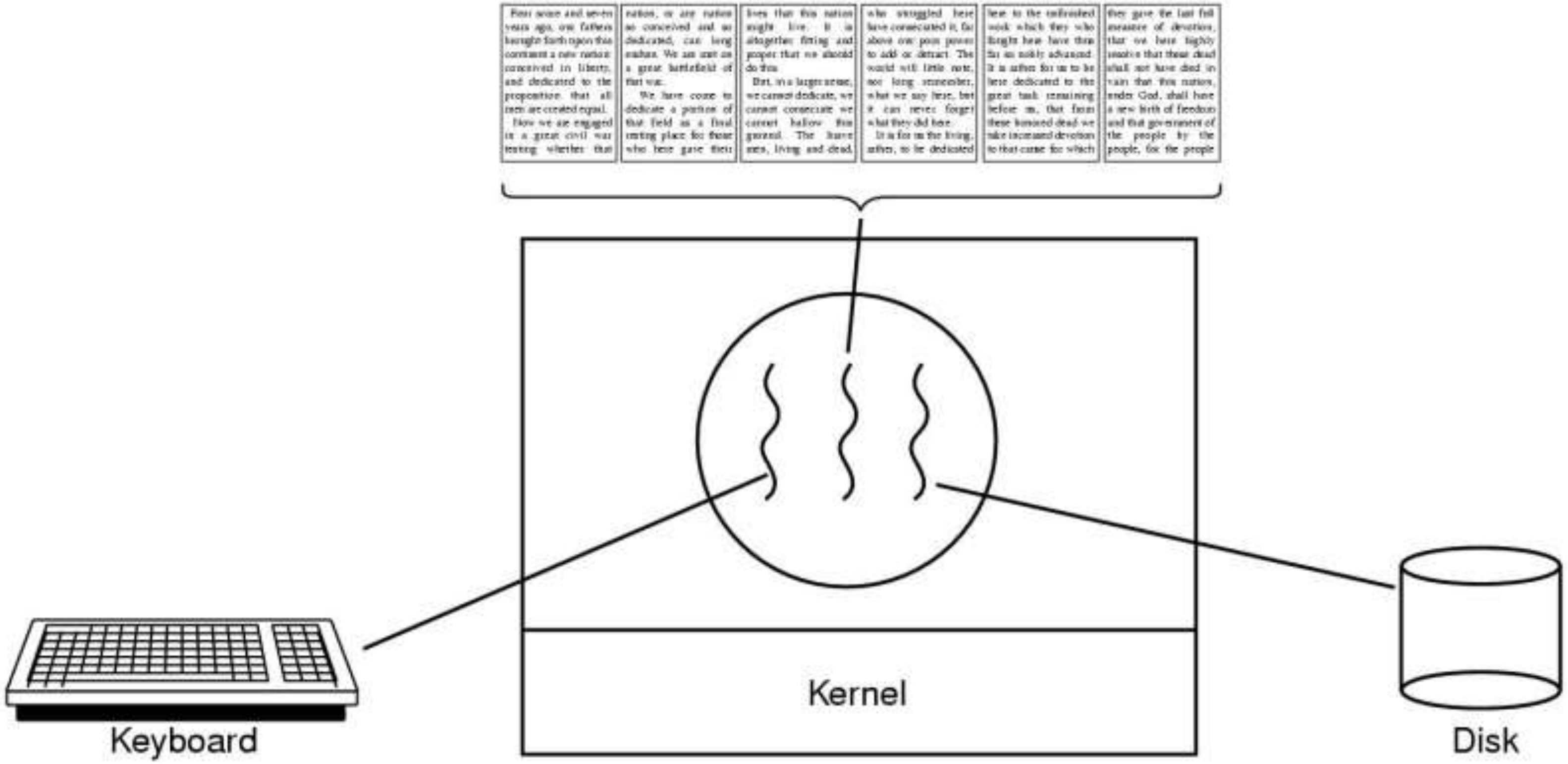
Alternative strategies for executing multiple rountines



Why Use Threads?

- **Utilize** multiple CPU’s concurrently
- **Low cost** communication via shared memory
- Overlap computation and blocking on a single CPU
  - *Blocking due to I/O*
  - *Computation and communication*
- Handle asynchronous events

Typical Thread Usage



Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.

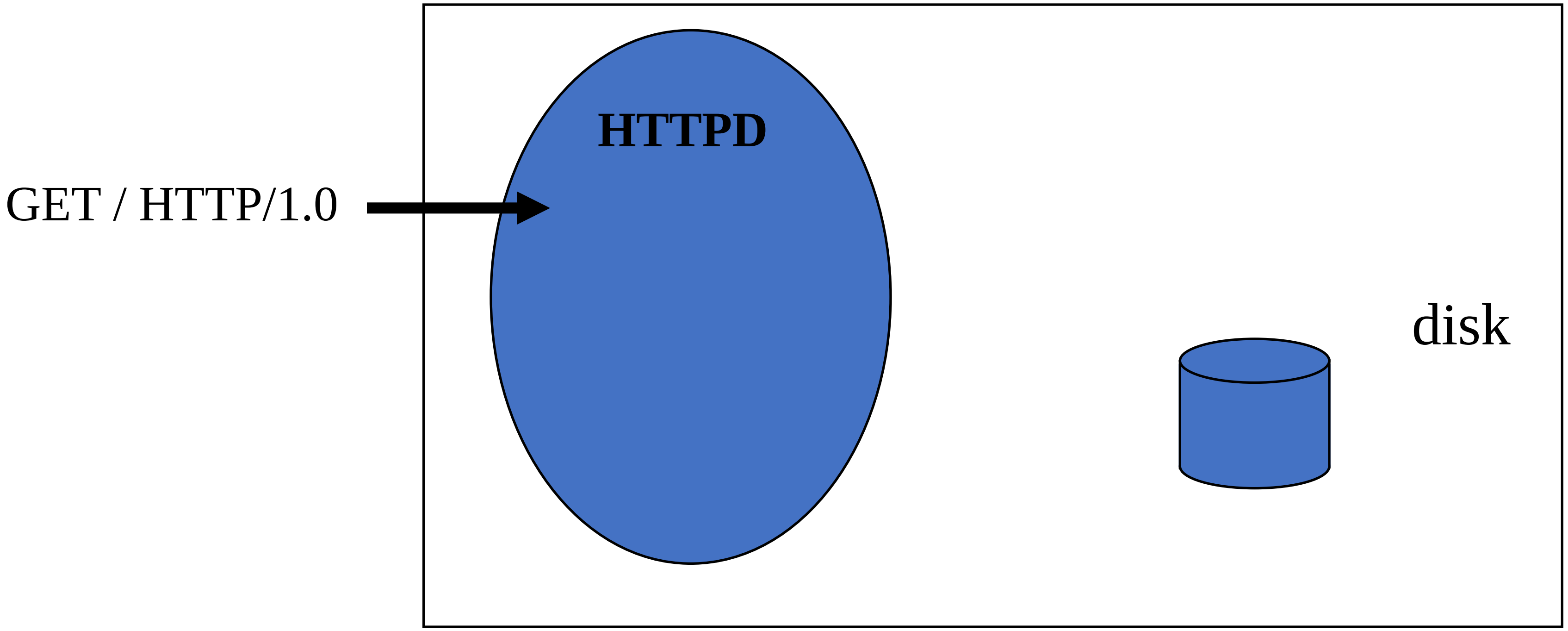
We have come to dedicate a portion of that field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.

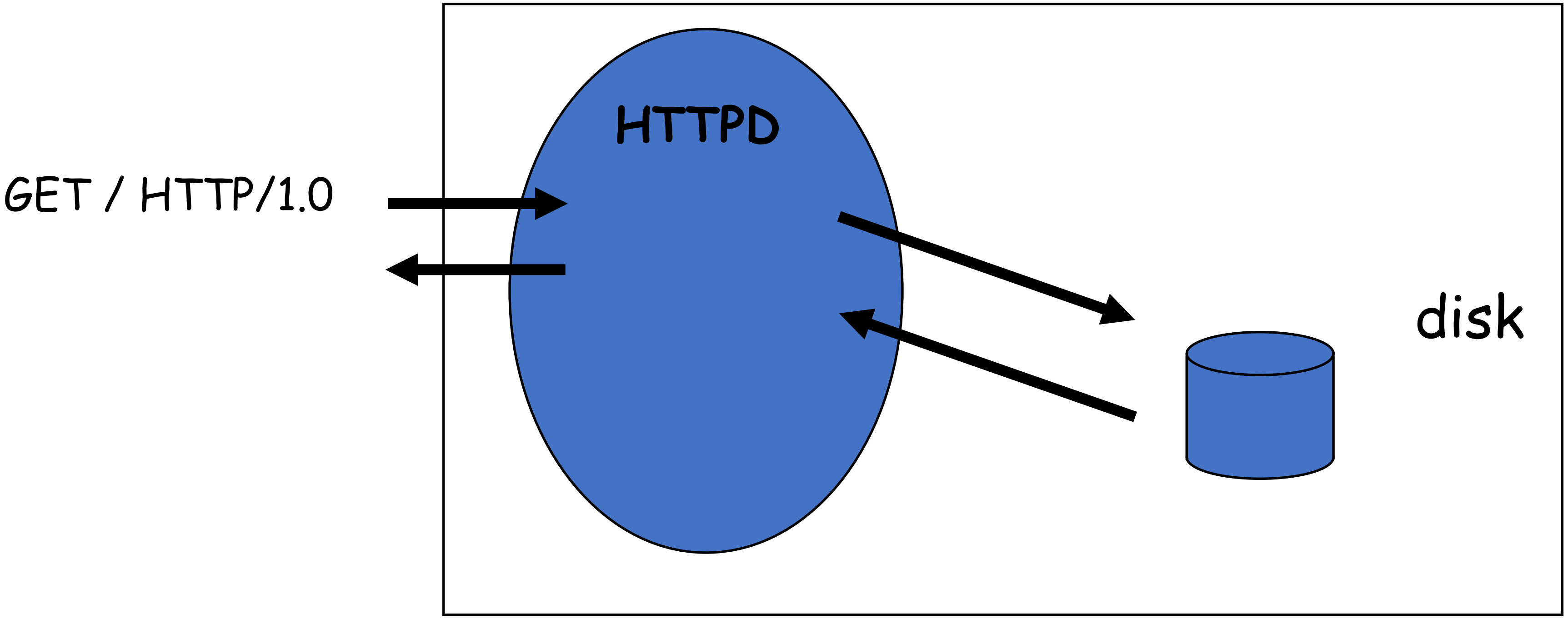
It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people, for the people

A word processor with three threads

Processes vs Threads

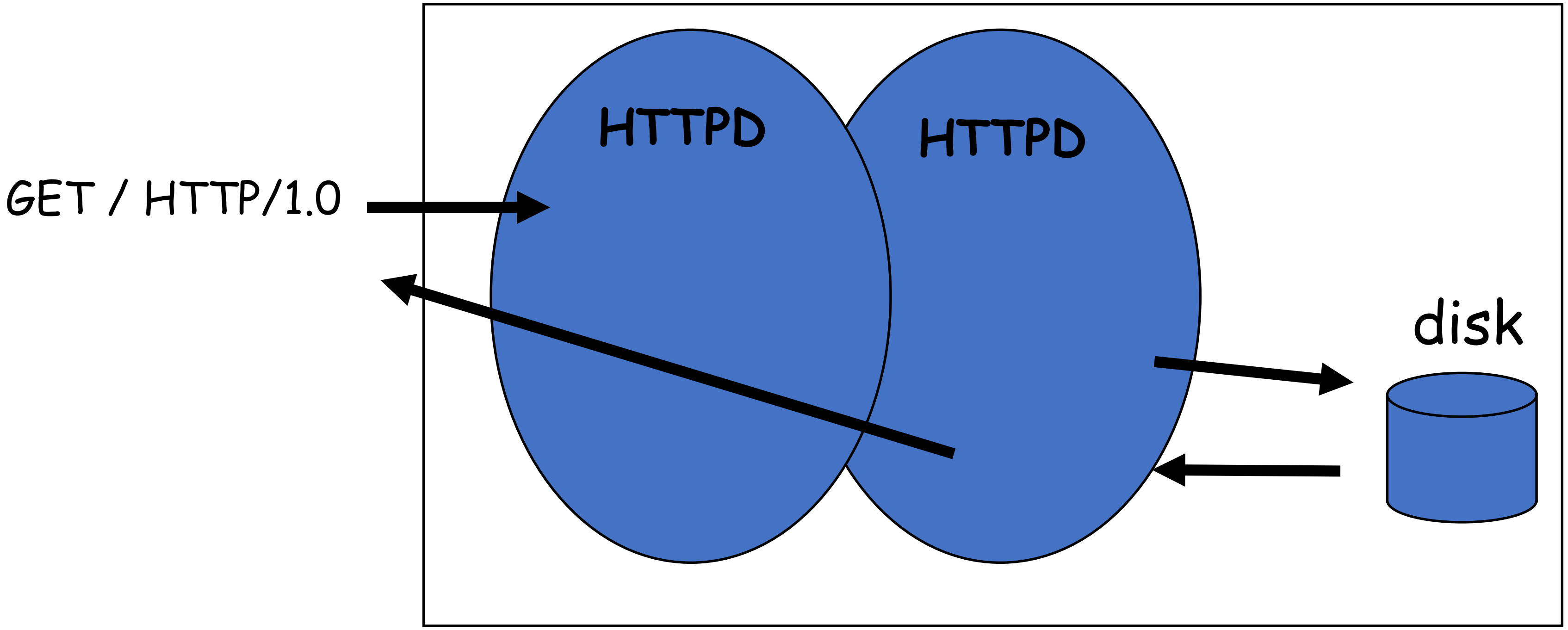


Processes vs Threads

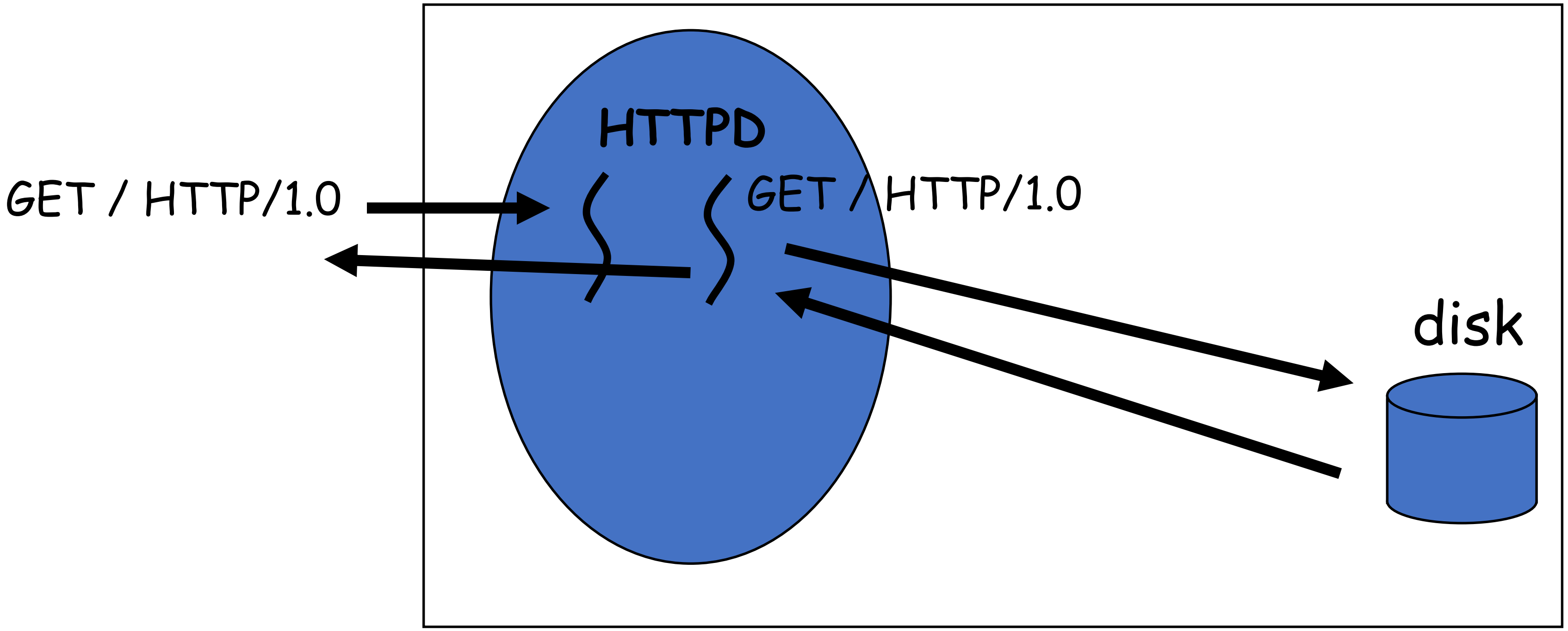


Why is this not a good web server design?

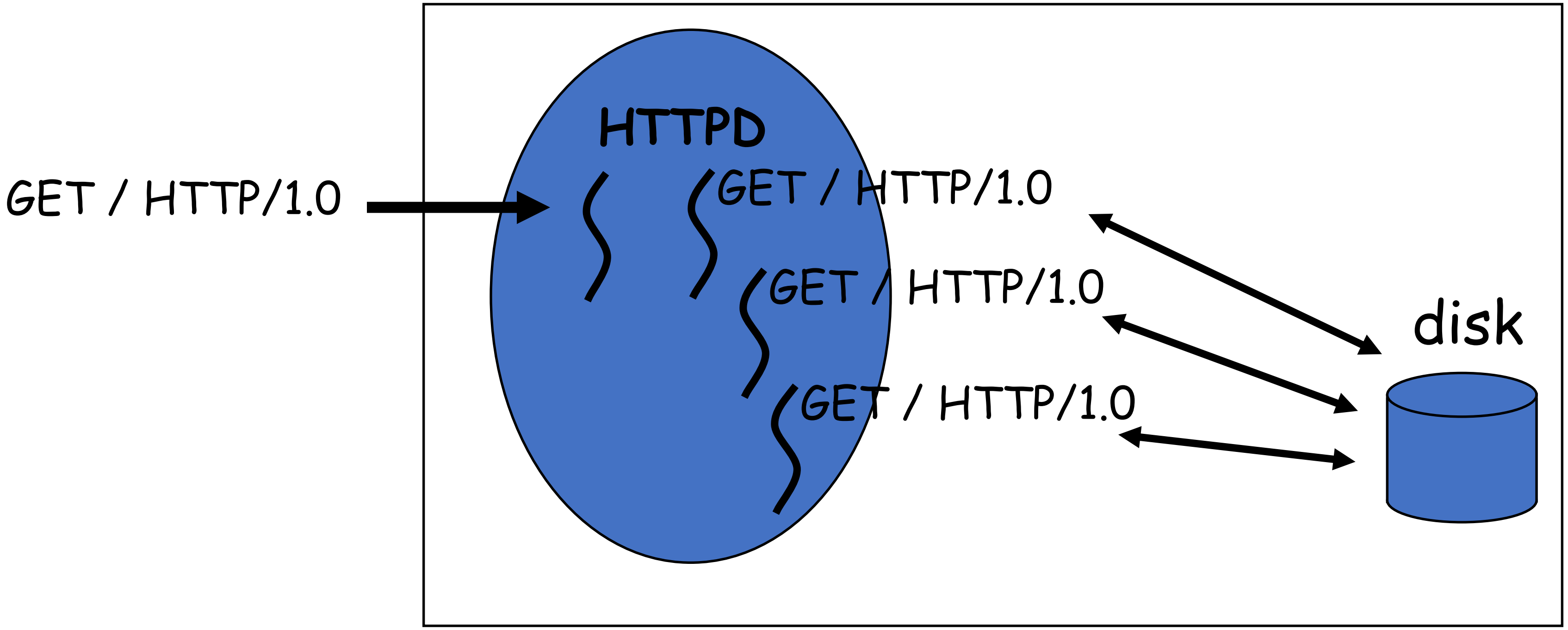
Processes vs Threads



Processes vs Threads



Processes vs Threads



GET / HTTP/1.0

HTTPD

GET / HTTP/1.0

GET / HTTP/1.0

GET / HTTP/1.0

disk

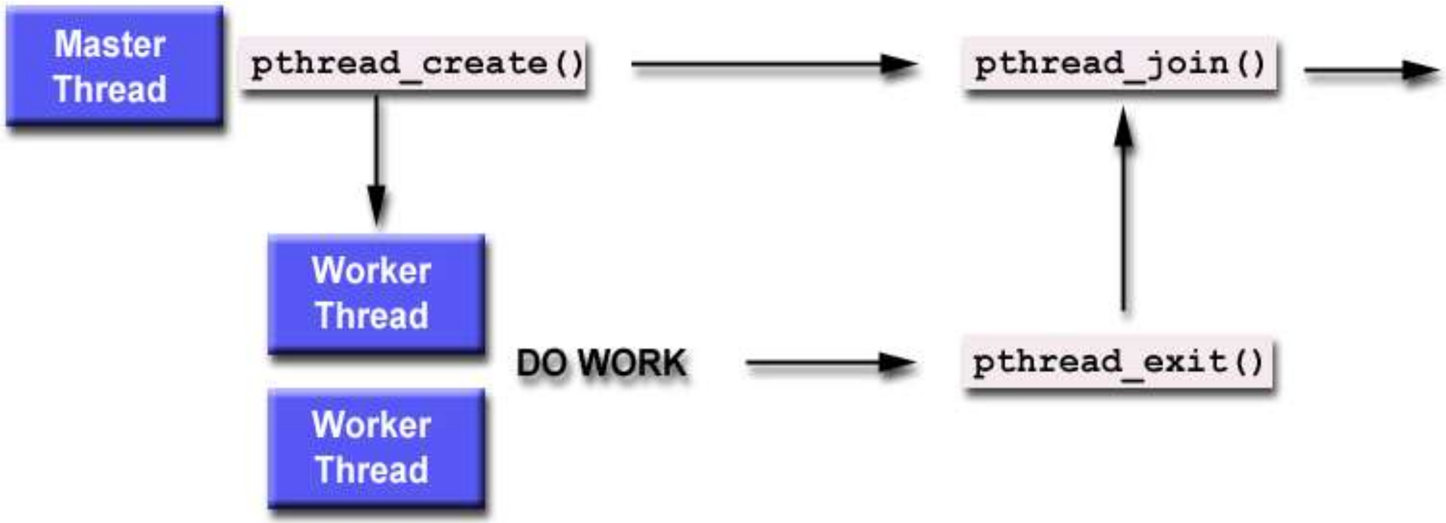
Common Thread Strategies

- **Manager/worker**
  - *Manager thread handles I/O*
  - *Magaer assigns work to worker threads*
  - *Worker threads created dynamically*
  - *... or allocated from a thread-pool*
- **Pipeline**
  - *Each thread handles a different stage of an assembly line*
  - *Threads hand work off to each other in a producer-consumer relationship*

Pthreads (continued)

- pthread_exit (status)
  - *Terminates the thread and returns "status" to any joining thread*
- pthread_join (threadid,status)
  - *Blocks the calling thread until thread specified by "threadid" terminates*
  - *Return status from pthread_exit is passed in "status"*
  - *One way of synchronizing between threads*
- pthread_yield ()
  - *Thread gives up the CPU and enters the run queue*

Using Create, Join and Exit



Pros & Cons of Threads

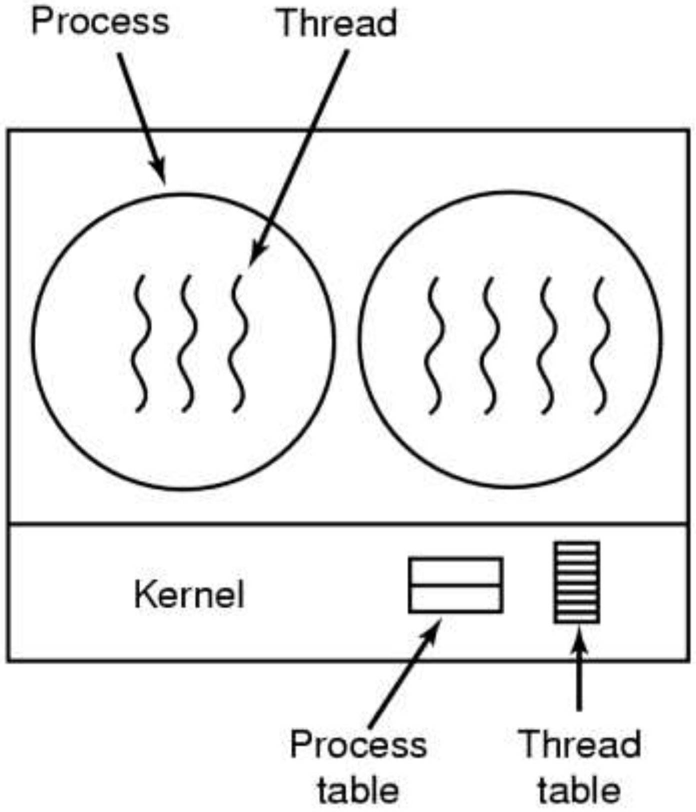
- Pros:
  - *Overlap I/O with computation!*
  - *Cheaper context switches*
  - *Better mapping to multiprocessors*
- Cons:
  - *Potential thread interactions*
  - *Complexity of debugging*
  - *Complexity of multi-threaded programming*
  - *Backwards compatibility with existing code*

User-level threads

- The idea of managing multiple abstract program counters above a single real one can be implemented using privileged or non-privileged code.
  - *Threads can be implemented in the OS or at user level*
- User level thread implementations
  - *Thread scheduler runs as user code (thread library)*
  - *Manages thread contexts in user space*
  - *The underlying OS sees only a traditional process above*

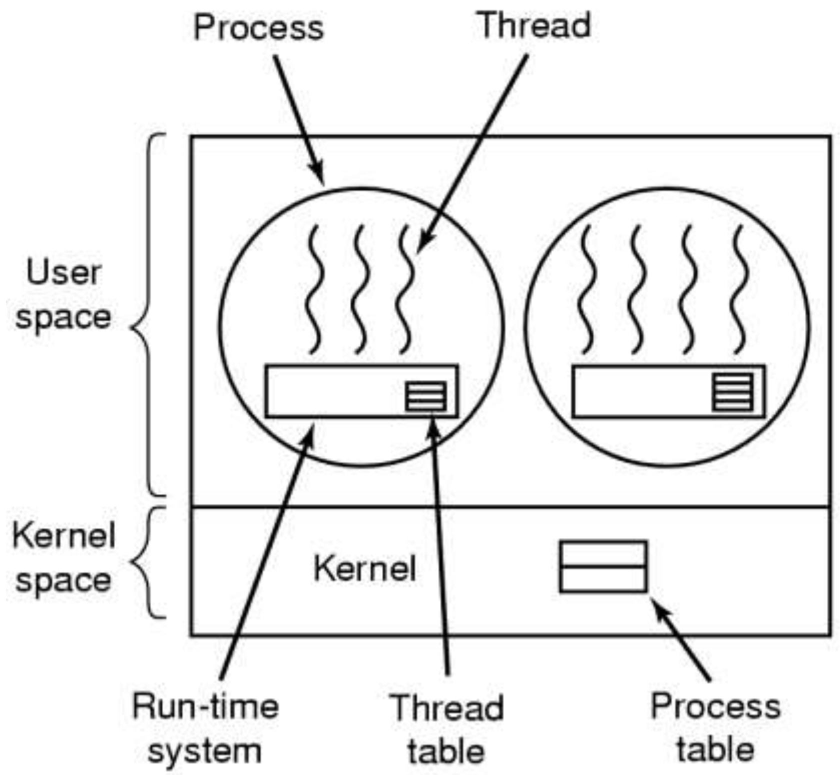
Kernel-Level Threads

Thread-switching code is in the kernel



Kernel-Level Threads

The thread-switching code is in user space



User-level threads

- Advantages
  - *Cheap context switch costs among threads in the same process!*
  - *Calls are procedure calls not system calls!*
  - *User-programmable scheduling policy*
- Disadvantages
  - *How to deal with blocking system calls!*
  - *How to overlap I/O and computation!*