Processes and Threads

Processes

- Non-determinism
- Why multiple pro
- Process creation, termination, switc
- Linux Case Study

Threads

- Concepts and models
- Threads vs processes
- Posix PThread case study
- Kernel and user threads

Introduction to Processes

- One of the oldest abstractions in computing
  - An abstraction of a running program
  - Encapsulates code and state of a program

- Allows a single process multiple programs “simultaneou
  - Processes turn a single CPU into multiple virtual CPUs
  - Each process runs on a virtual CPU

Why Have Processes?

Provide (the illusion of) concurrency

- Real vs. apparent concurrency

Provide isolation

- Each process has its own address space

Simplicity of p

- E.g. Firefox d bout gcc

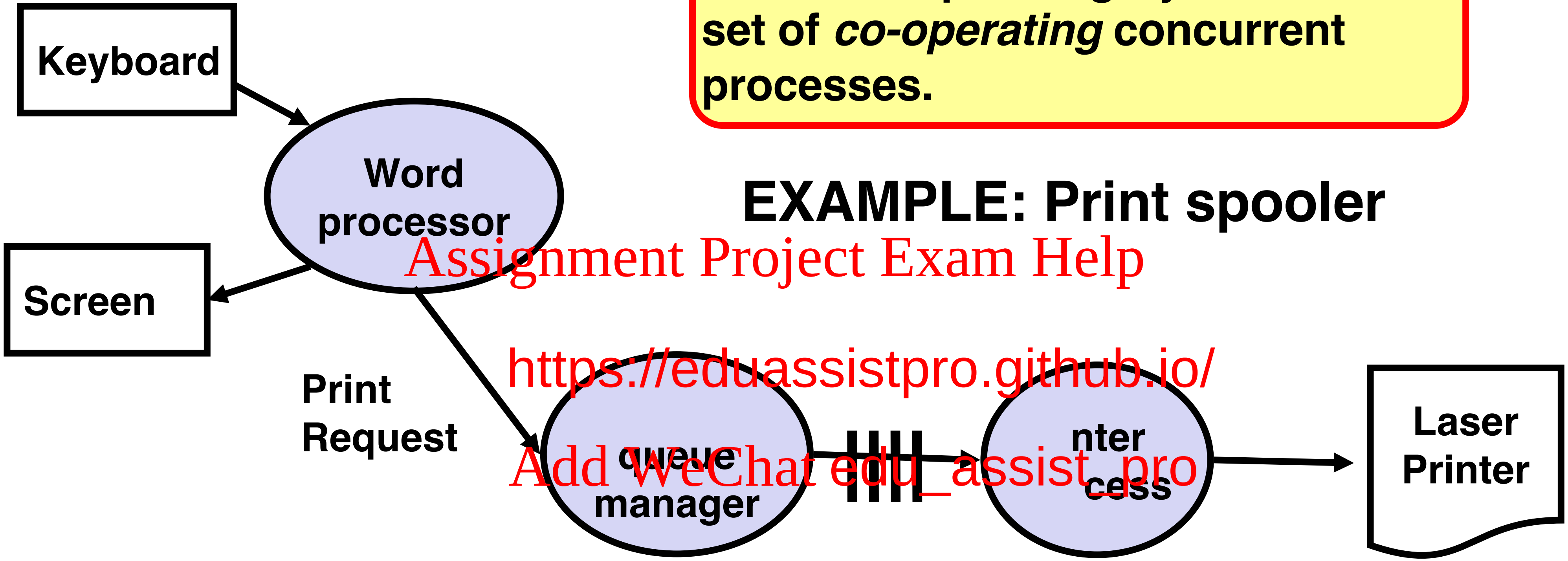
Allow better utilization of esources

- Different processes require different resources at a certain time

Processes for OS Structuring

Consider Operating System as a set of *co-operating* concurrent processes.

EXAMPLE: Print spooler



Keyboard & screen: processes to manage these devices
Word processor: User edits document, requests printing
Print queue manager: Maintains queue of jobs for printer. If queue was previously empty, starts printer process.
Printer Process: Translates document to printer commands, and sends them to it. On completion, removes job from queue, and repeats. Terminates when queue is empty.

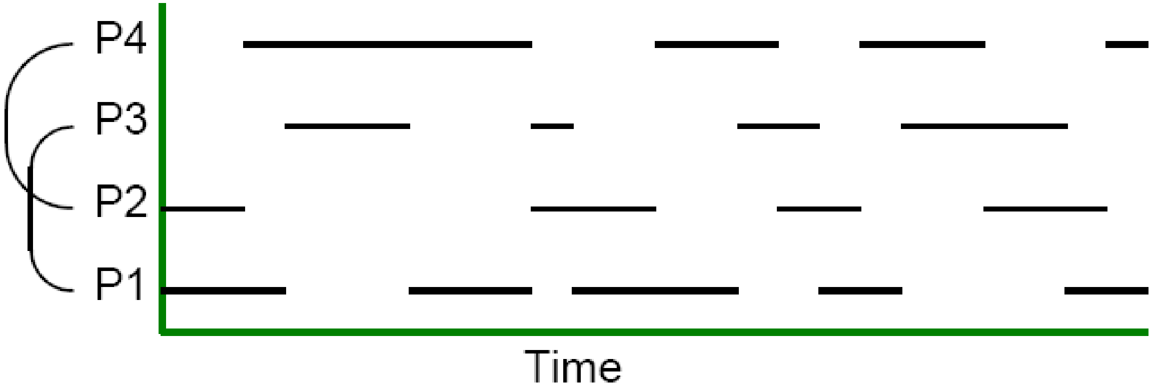
Non - Determinism

- Operating Systems and Real-Time systems are **non-deterministic**
- They must respond to events (I/O) which occur in an unpredictable order, and at any time

Concurrency

- **Apparent Concurrency (pseudo-concurrency):** A single hardware processor which is switched between processes by interleaving. Over a period of time this gives the illusion of concurrent execution.

- **Real Concurrency**: Multiple hardware processors; usually less processors than processes



Process Switches

Events (or interrupts) cause process switches.

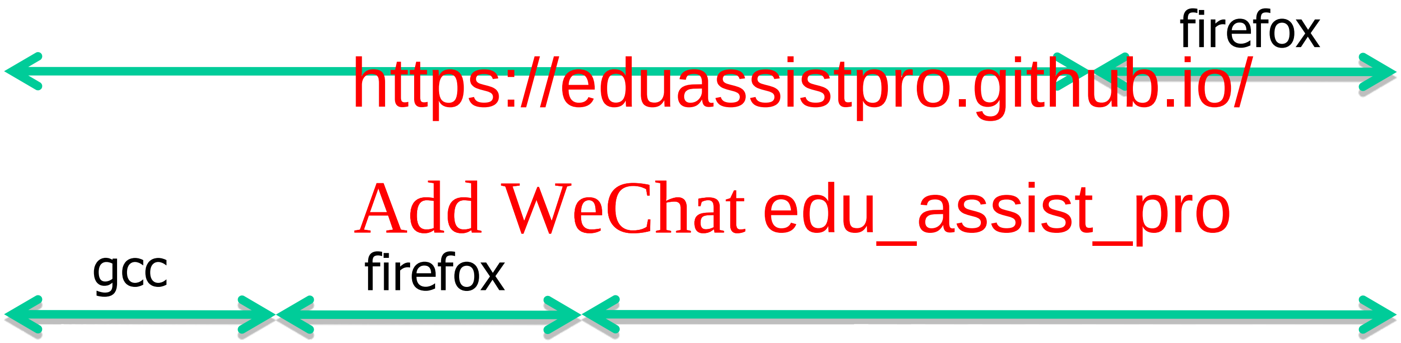
- For example, an I/O completion interrupt will cause the OS to switch to an I/O process

The way an OS switches between processes cannot be pre-determined, cause the switches are not

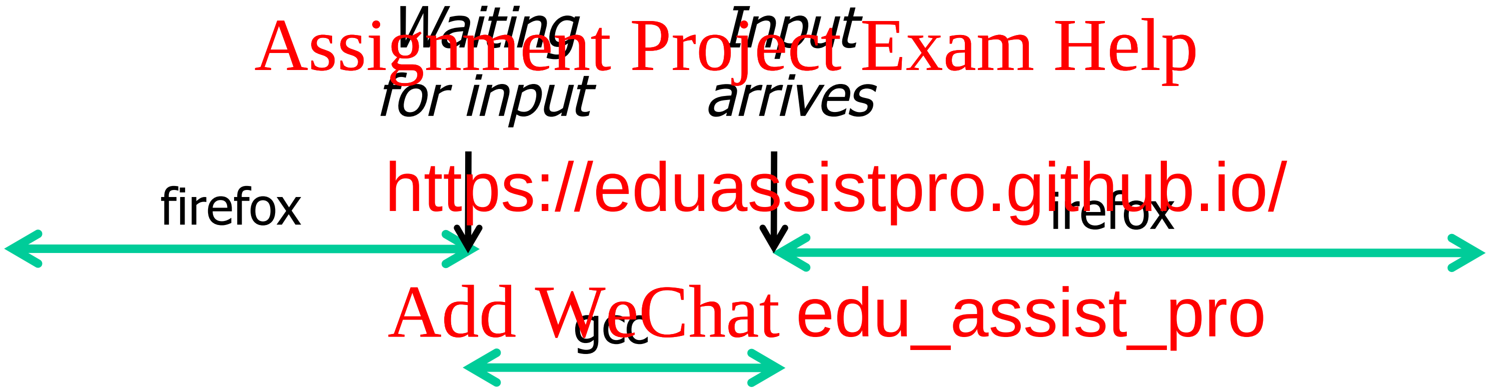
The interleaving of instructions, executed by a processor, from a set of processes is non-deterministic

- Not reproducible, no built-in assumptions about timing

Fairness



Better CPU utilization



menti.com Multiprogramming Q1 98 63 88

Why Multiprogramming?

Why do most Operating Systems provide multiprogramming?

CPU Utilization in Multiprogramming

Q: Average process computes 20% time, then with five processes we should have 100% CPU utilization, right?

A: In the ideal case, if the five processes never wait for I/O at the same time

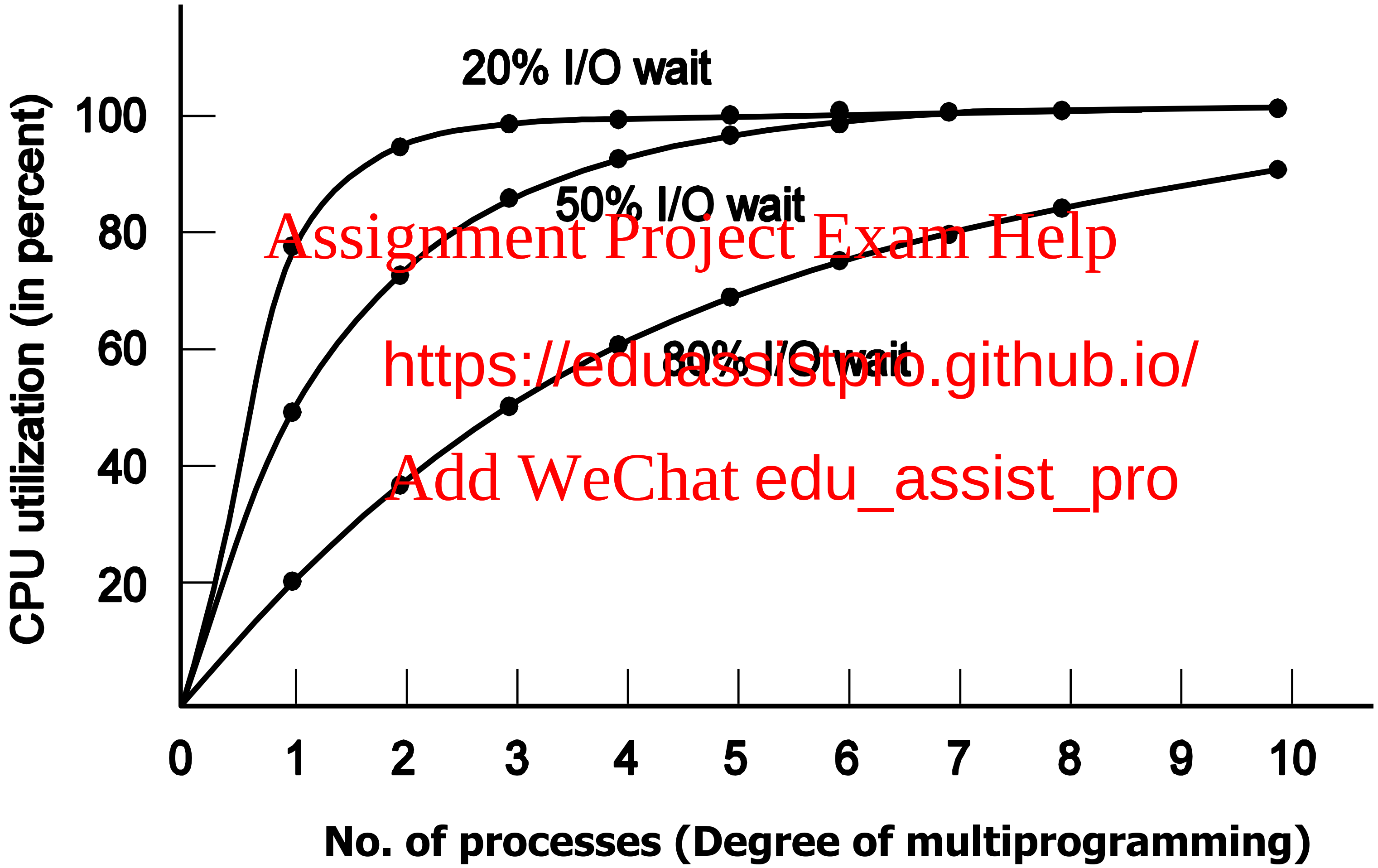
- Better estimate
  - **n** = total nu
  - **p** = fraction of time a proces or I/O
  - **p**^{n} = probability that all proce ting for I/O

$$\texttt{CPU utilization} = 1 - p^n$$

Q: How many processes need to be in memory to only waste 10% of CPU where we know that processes spend 80% waiting for I/O (e.g. data oriented or interactive systems)?

menti.com Q2 CPU utilization 98 63 88

CPU Utilization = 1 – p^n



Context Switches

On a context switch, the processor switches from executing process A to executing process B, because:

- Time slice expired (periodic)
- Process A blocked waiting for e.g. I/O or a resource
- Process A completed (run to completion)
- External even [illegible] rity process B to be run (priority p [illegible]

Non-deterministic process switches as events causing them are non-deterministic.

Context Switches

On a context switch, the processor switches from executing process A to executing process B

Process A may be restarted later, therefore, all information concerning the process, needed to restart safely, should be

For each process in a *process descriptor*, or *process control*), which is kept in the *process table*

Process Control Block (PCB)

A process has its own virtual machine, e.g.:

- Its own virtual CPU
- Its own address space (stack, heap, text, data etc.)
- Open file descriptors, etc.

What state information should be stored?

- Program coun[illegible]ister, stack pointer, etc.

- Process management info:
  - Process ID (PID), parent process, process group, priority, CPU used, etc.
- File management info
  - Root directory, working directory, open file descriptors, etc.

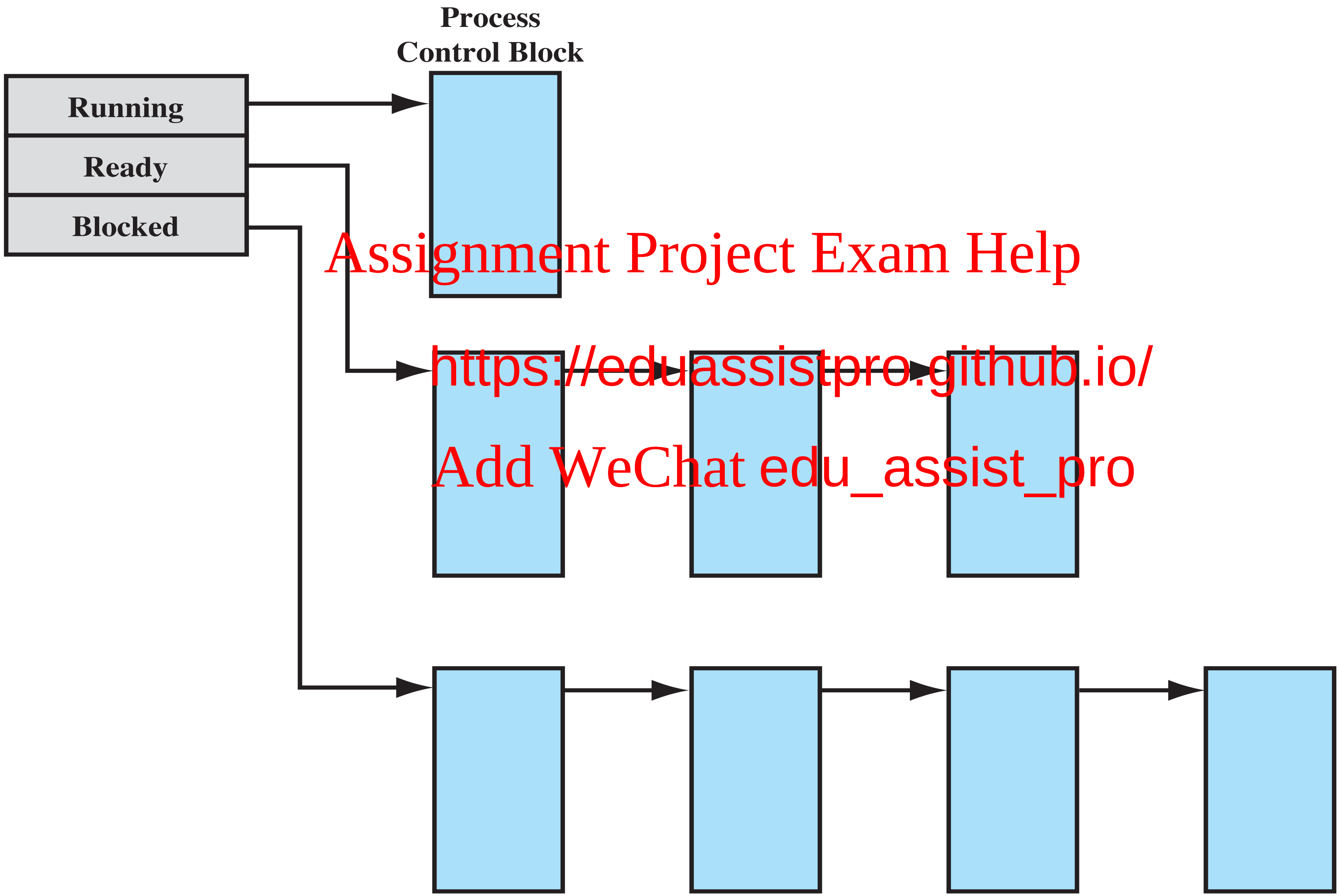
Simplified Process Control Block (PCB)

PCB: Data structure representing a process in the kernel

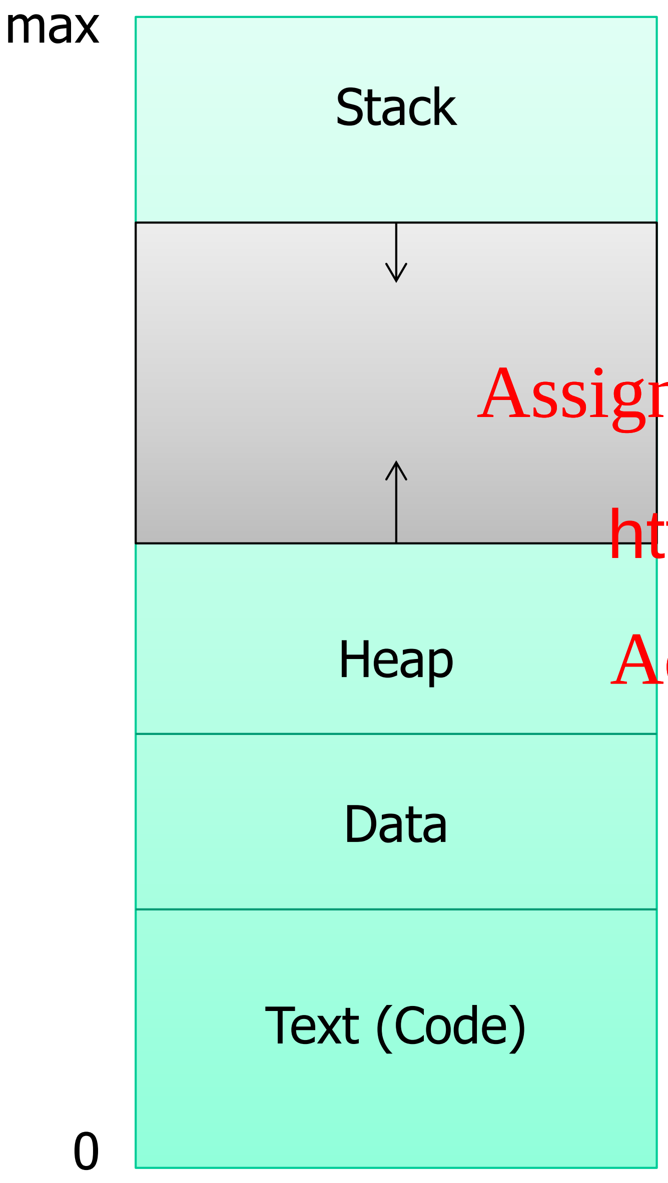
- **Process IDs:** unique identifier to distinguish it from other processes.
- **State:** running, waiting, ready etc. (details later)
- **Priority:** priority level relative to other processes
- **Program cou** ction in program to be executed
- **Context data:** data saved fro
- **Memory pointers:** to progra associated with process and shared memory with other processes
- **I/O status:** I/O requests outstanding, I/O devices allocated
- **File Management:** Required directories, list of open files
- **Accounting information:** processor time used, time limits, memory limits, file usage + limits etc

Detailed PCB

Process List Structures



Process in Memory



Stack: temporary data e.g. function parameters, return addresses, local variables.

Heap: dynamically allocated data structures.

riables.

Text: de ocesses can share code e.g. 2 concurrent word processors can edit different files, using same code.

Process Switch Implementation

1. Each IO class has interrupt vector containing the address of interrupt service procedure
2. On interrupt the PC, PSW, some registers p the (current) sta *interrupt hardware*
3. Hardware jumps to address (PC from Interrupt vector) to service interrupt
4. Assembly language routine saves registers to PCB then calls device specific interrupt service routine
5. C interrupt service runs (typically reads, writes & buffers data)
6. ***Scheduler*** decides which process to run next

edure returns

l to assembly code

y procedure starts current process

Context (Process) Switches are Expensive

Direct cost: save/restore process state

Indirect cost: perturbation of memory caches, memory management registers etc.

Important to av t switches

Process Creation

When are processes created?

- System initialisation
- User request
- System call by a running process

Processes can b

- Foreground p sers
- Background processes: hand mail, printing requests, etc. (**daemons**)

Process Termination

- **Normal completion:** Process completes execution of body
- **System call:**
  - `exit()` in UNIX
  - `ExitProcess()` in Windows

- **Abnormal exi** into an error or an unhandled exce uction, memory violation

- **Aborted:** The process stops nother process has overruled its execution (e.g., killed from terminal)
- **Never:** Many real-time processes run in endless loop and never terminate unless error occurs

Process Hierarchies

Some OSes (e.g., UNIX) allow processes to create **process hierarchies** e.g. parent, child, child’s child, etc.

- E.g., when UNIX boots it starts running **`init`**
- It reads a file saying how many terminals to run, and forks off one process per terminal
- They wait for s
- When login suc tes a shell to accept commands which in turn may s processes etc.
- All processes in the entire system form a process tree with **init** as the root (***process group***)

Windows has no notion of hierarchy

- When a child process is created the parent is given a token (***handle***) to use to control it
- The handle can be passed to other processes thus no hierarchy

Hardware Support for Multiprogramming

Explain why multiprogramming systems require:

a) Hardware interrupts from I/O devices

b) Independent direct memory access channel

Cas inux

Creating processes

```
int fork(void)
```

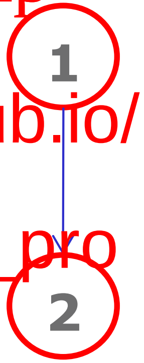
- Creates a new child process by making an exact copy of the parent process image.
- The child process inherits the resources of the parent process and will be execu parent process.
- **fork()** returns t
  - In the parent process: **fork()** process ID of the child
  - In the child process: **fork()** returns 0
- On error, no child is created and -1 is returned in the parent
- How can fork() fail?
  - Global process limit exceeded, per-user limit exceeded, not enough swap space

fork() example (1)

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

int main() {
  if (fork()
    printf("Parent code\
  else printf("Child cod

  printf("Common code\n");
}
```



"Parent code"
"Common code"

"Child code"
"Common code"

fork() example (2)

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

int main() {
  if (fork() != 0)
    printf("X

  if (fork() != 0)
    printf("Y\n");

  printf("Z\n");
}
```

menti.com Q3: 98 63 88

What does initial process print?

Executing processes

```
int execve(const char *path, char *const argv[],
           char *const envp[])
```

Arguments:

- **path** – full pathname of program to run

- **argv** – argum

- **envp** – envir PATH, $HOME)

Changes process image and ru ocess

Lots of useful wrappers:

E.g., execl, execle, execvp, execv, etc.

```
man execve
```

Consult man(ual) pages!

Waiting for Process Termination

```
int waitpid(int pid, int* stat, int options)
```

- Suspends execution of the calling process until the process with PID pid terminates normally or a signal is received

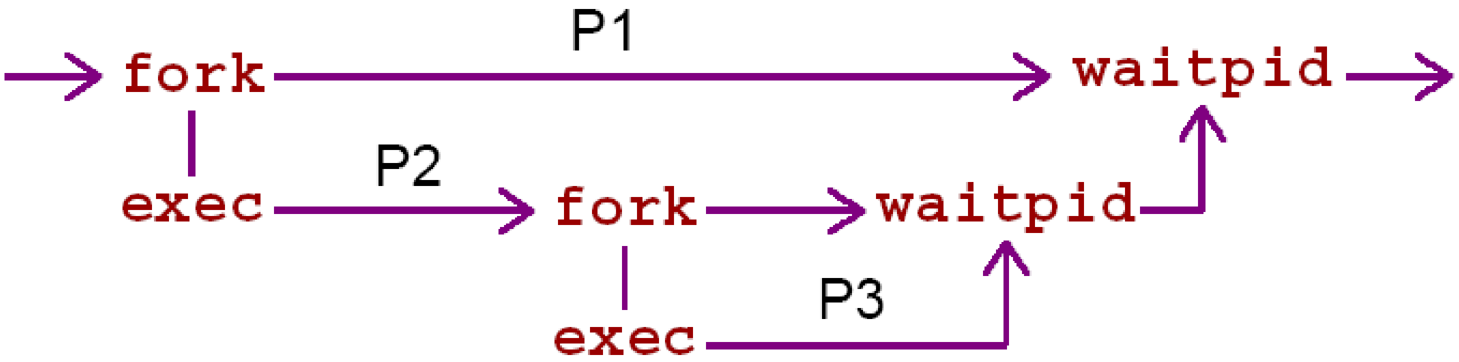
- Can wait for m

  - pid = -1 wait
  - pid = 0 wait for any child in th ess group as caller

  - pid = -gid wait for any child with process group gid
- Returns:
  - pid of the terminated child process
  - 0 if WNOHANG is set in options (indicating the call should not block) and there are no terminated children
  - -1 on error, with errno set to indicate the error

Example: Command Interpreter

Use of `fork`, `execve` and `waitpid`



P1

fork

waitpid

exec

P2

fork

waitpid

exec

P3

Why both fork() and execve() ?

UNIX design philosophy: **simplicity**

- Simple basic blocks that can be easily combined

Contrast with Windows:

- CreateProcess() => equivalent of fork() + execve()
- Call has 10 parameters!
  - program to
  - parameters
  - security attributes
  - meta data regarding files
  - priority,
  - pointer to the structure in which info regarding new process is stored and communicated to the caller
  - …

Windows CreateProcess()

```
BOOL WINAPI CreateProcess(
    __in_opt LPCTSTR lpApplicationName,
    __inout_opt L
    __in_opt LPSE                   ocessAttributes          ,
    __in_opt LPSECURITY_ATTRIBU               adAttributes,
    __in BOOL bInheritHandles,
    __in DWORD dwCreationFlags,
    __in_opt LPVOID lpEnvironment,
    __in_opt LPCTSTR lpCurrentDirectory,
    __in LPSTARTUPINFO lpStartupInfo,
    __out LPPROCESS_INFORMATION lpProcessInformation )
```

Linux Termination

```
void exit(int status)
```

- Terminates a process
- Called implicitly when program finishes execution
- Never returns in the calling process
- Returns an ex

```
void kill(int pid, int sig)
```

- Sends signal sig to process pid to terminate it.

Threads

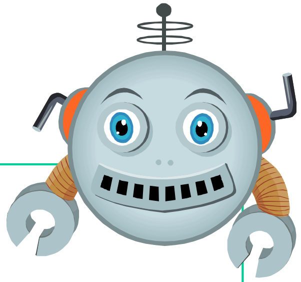
What Are Threads?

- Execution streams that share *the same address space*
- When multithreading is used, each process can contain one or more threads
  - a *lightweight mini-process* within a user process

3 Processes, each with 1 thread

1 process with 3 threads

One or More Threads in a Process



Each thread has:

- an executi ady, etc.)
- saved thread context w ning
- an execution stack
- some per-thread static storage for local variables
- access to the memory and resources of its process (all threads of a process share this)

Thread Model

Per process items	**Per thread items**
Address space	Program counter (PC)
Global variables	
Open files	
Child processes	Stat
Pending alarms	
Signals and signal handlers	
Accounting information	

Thread Model (2)

Each thread has its own stack & context

Registers for Threads

The register set is a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.

Example Word Processor

Processing thread

- processes input buffer
- writes result into output buffer

Input thread

- reads data into buffer

Output thread

- writes output buffer to disk

Example Multi-threaded Web Server

Threads vs Proceses

Processes are too heavyweight

- Expensive to create/destroy activities
- Difficult to communicate between different address spaces
- An activity that might switch out the entire application
- Expensive to context switch between activities

Threads are lightweight

- Create/delete up to 100 times quicker
- Activities can share data

cient communication
ween threads

t parallelism
application,
where some activities may block

Threads – Problems/Concerns

- Shared address space
  - Memory corruption
    - One thread can write another thread’s stack
  - Concurrency bugs
    - Concurrent access to shared data (e.g., global variables)

- Forking
  - What happen

    - Create a new process with t ber of threads

    - Create a new process with a d?
      - Single thread i.e. the thread which executed fork
- Signals
  - When a signal arrives, which thread should handle it?
    - For fault, the thread causing the fault
    - For other signal e.g. SIGALARM, any thread

Case Study: PThreads

PThreads (Posix Threads)

Defined by IEEE standard 1003.1c

- Implemented by most UNIX systems

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

pthread_t → type representing a thread

pthread_attr_t → type representing the attributes of a thread

Creating Threads

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

Creates a new thread

- The newly created thread is stored in ***thread**

- The function return ully created, or error code

Arguments:

- **attr** -> specifies thread attribute **LL** for default attributes
  - Attributes include: minimum stack size, guard size, detached/ joinable, etc.
- **start_routine** -> the C function the thread will start to execute once created
- **arg** -> The argument to be passed to start_routine (of pointer type **void***). Can be **NULL** if no arguments are to be passed.

Terminating Threads

```
void pthread_exit(void *value_ptr);
```

Terminates the thread and makes **value_ptr** available to any successful joi thread

Called implicitly w routine returns

- But not for the initial thread d main()
- If **main()** terminates before ds, w/o calling **pthread_exit()**, the entire process is terminated
- If **pthread_exit()** is called in **main()** the process continues executing until the last thread terminates (or **exit()** is called)

PThread Example

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

void *thread_work(void *thread_id) {
  long id = (long) thr
  printf("Thread %ld\n
}

int main (int argc, char *argv[]) {
  pthread_t threads[5];
  long t;
  for (t=0; t<5; t++)
      pthread_create(&threads[t], NULL,
                     thread_work, (void *)t);

}
```

```
$ gcc pt.c -lpthread
$ ./a.out
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
$ ./a.out
Thread 0
Thread 3
Thread 1
Thread 2
```

Passing Arguments to Threads

What if we want to pass more than one argument to the start routine?

- Create a structure containing the arguments and pass a pointer to that structure to `pthread_create()`

Yielding the CPU

```
int pthread_yield(void)
```

- Releases the CPU to let another thread run
- Returns 0 on suc
  - Always succeed

Why would a thread ever voluntarily give up the CPU by calling thread yield()?
After all, since there is no periodic clock interrupts, it may never get the CPU back.

Joining Other Threads

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

Blocks until **thread** terminates

The value passe by the terminating thread is availabl nced by **value_ptr**

- **value_ptr** can be **NULL**

Join Example

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

long a, b, c;
void *work1(void *x) { a = (long)x *
  (long)x;}
void *work2(void *y) { b = (long)y *
  (long)y;}

int main (int argc, char *argv[]) {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, work1, (void*)
 3);
  pthread_create(&t2, NULL, work2, (void*)
 4);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  c = a + b;
  printf("3^2 + 4^2 = %ld\n", c);
}
```

```
$ ./a.out
3^2 + 4^2 = 25
```

Threads Implementation

User-level threads

- The kernel is not aware of threads
- Each process manages its own threads

Kernel-level t

- Managed by the kernel

Hybrid

- Combined Kernel and user level threads
- User threads map onto kernel threads

User-Level Threads

- Kernel thinks it is managing processes only
- Threads implemented by software library

- Thread switc ernel mode privileges

- Process maintains a thread does thread scheduling

- PThread is user level

USER Level Threads

Advantages of User-Level Threads

Better performance

- Thread creation and termination are fast
- Thread switching is fast
- Thread synchronization (e.g., joining other threads) is fast
- All these oper y kernel activity

Allows application-specific run

- Each application can have its own scheduling algorithm

Disadvantages of User-Level Threads

Blocking system calls stops *all threads* in the process

- Denies one of the core motivations for using threads

Non-blocking I/O can be used (e.g., **`select()`**)

- Harder to use and understand, inelegant

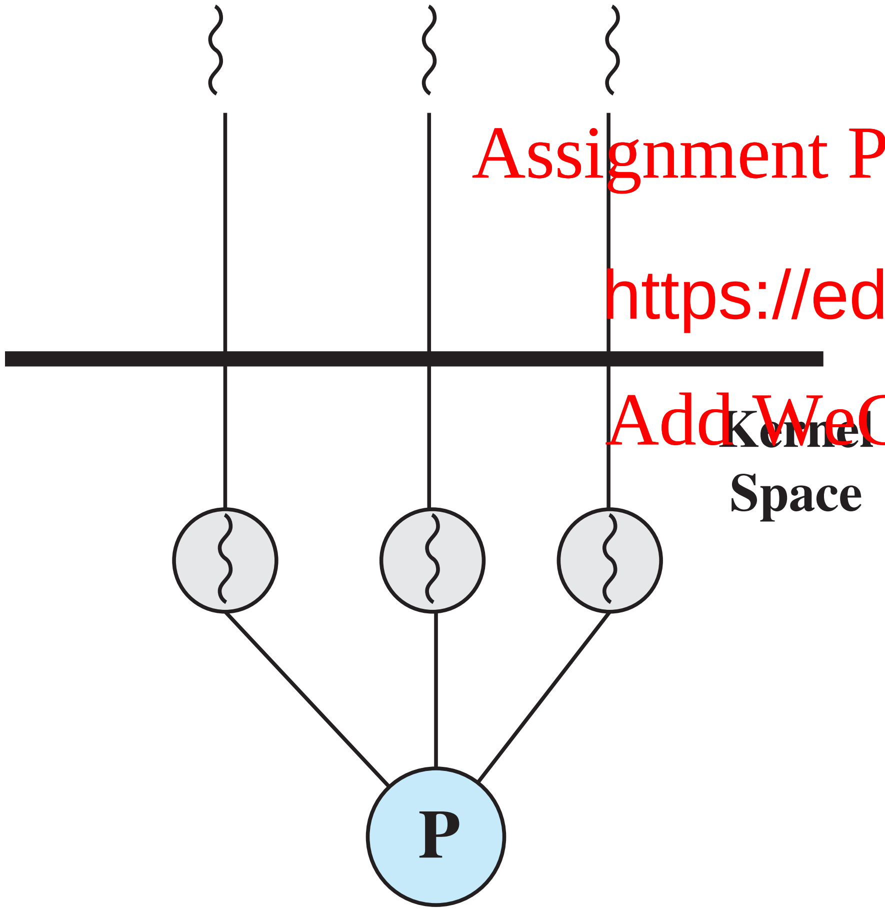
During a page fa hole process…

- But other threads might be r

Difficult to implement preemptive scheduling

- Run-time can request a clock interrupt
    - Messy to program
    - High-frequency clock interrupts not always available
    - Individual threads may also need to use a clock interrupt

Kernel Threads



Thread management is done by the kernel

- no thread
  nagement is done
  the application
  ows is an
  ple of this
  approach
- Recent Linux implementations also support this.

Advantages of Kernel Threads

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- Blocking system calls/page faults can be easily accommodated
  - If one thread call or causes a page fault, the ker ble thread from the same process
- Kernel routines can be multithreaded

Disadvantages of Kernel Threads

Thread creation and termination more expensive

- Require kernel call
- But still much cheaper than process creation/termination
- One mitigation strategy is to recycle threads (*thread pools*)

Thread synchron e

- Requires blocking system cal

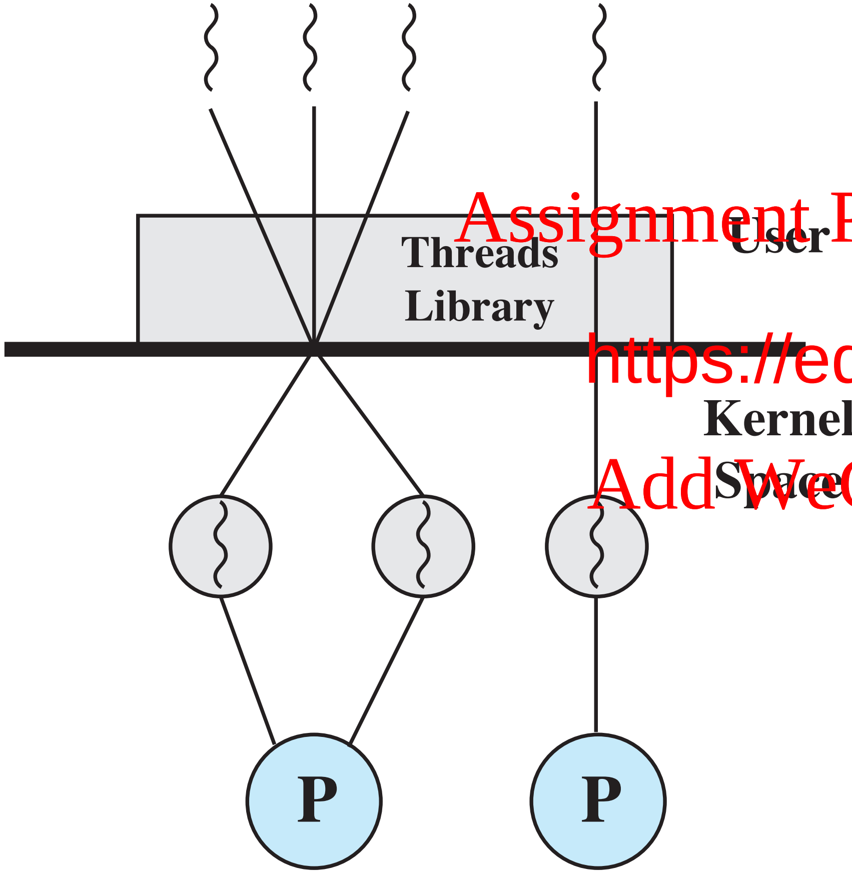
Thread switching is more expe

- Requires kernel call
- But still much cheaper than process switches
  - Same address space

No application-specific scheduler

menti.com Q 4 Stacks 98 63 88

Hybrid Approaches



- Thread creation is done in the user space
- Use kernel threads multiplex user-threads onto or all) kernel s
- Bulk of scheduling and synchronization of threads is by the application

Multithreaded Web Server

If in a multithreaded web server the only way to read from a file is the normal blocking read() system call, do you think user-level threads or kernel-level threads are being used? Why?

Process and Thread Summary

Non-determinism ➔ concurrency ➔ multiple processes ➔ better utilization

Processes: creation, termination, switching & PCBs

- Heavyweight management

Linux – supports process hierarchies

- Child is clone

- Load new code to execute d cess

Threads: lightweight concurre hared data

Posix threads case study

Thread implementation – user vs kernel level

Shared memory in threads requires synchronisation

Thread switching can be controlled by programmer

When Do Threads Improve Efficiency?

Would an algorithm that performs several independent CPU-intensive calculations concurrently (e.g., matrix multiplication) be more efficient if it used threads, or if it did not use threads?

Hint: consider uniprocessor and multiprocessor

Menti.com: Q5 Thread efficien 8