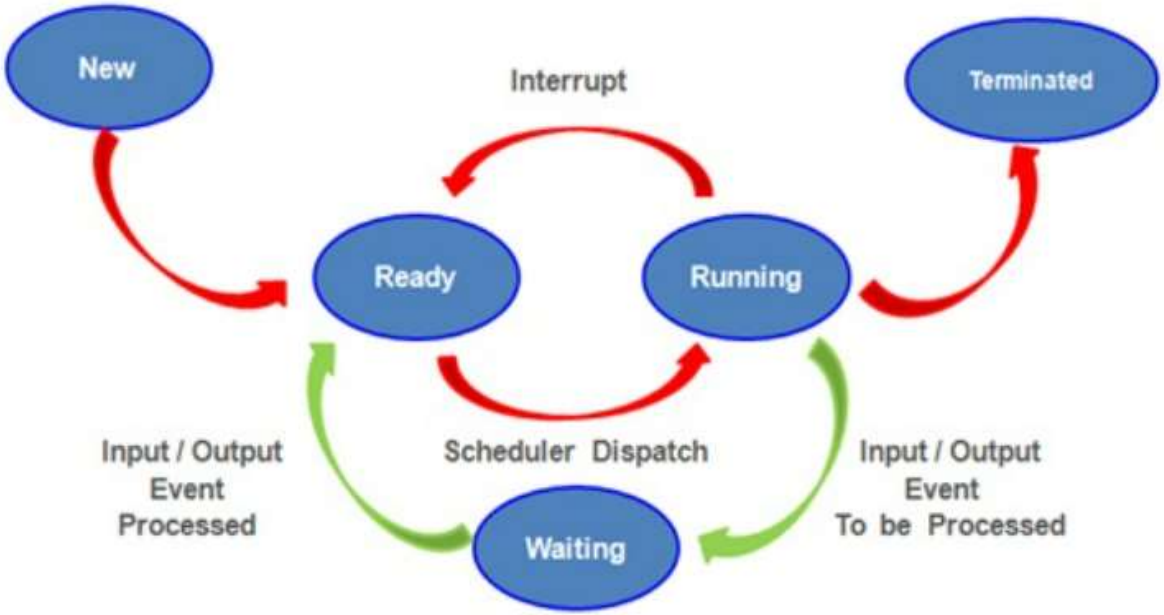


Chapter 2 Processes and Threads

CSS 226
Operating Systems

Chukiat Worasucheep

Important Notice

การเรียนการสอนหัวข้อนี้ ผ่านทางสื่อออนไลน์ (Online meeting)

และมีการบันทึกภาพและเสียงเพื่อประโยชน์ทางการศึกษาต่อไปในอนาคต.

หากท่านไม่ยินยอมให้มีการเผยแพร่การบันทึกดังกล่าว ขอให้แจ้งให้ผู้สอนทราบภายใน 36 ชั่วโมง.

(พรบ. พระราชบัญญัติคุ้มครองข้อมูลส่วนบุคคล (PDPA) พ.ศ.2562)

Course

1. บทนำ (Definition, Goals, Kernel, Structures and types of OS's)
2. กระบวนการและสายงาน (Processes & Threads)
3. การจัดตารางทำงานกระบวนการ (Process Scheduling)
4. การควบคุมภาวะพร้อมกัน (Concurrency control)
5. การจัดการหน่วยความจำ (Memory management)
6. หน่วยความจำเสมือน (Virtual memory)
7. ระบบอุปกรณ์เข้าออกและแหล่งเก็บข้อมูล (I/O systems and storages)
8. ระบบความมั่นคงคอมพิวเตอร์ (Computer security)

Process Management

Memory Management

I/O and Storage Management

Learning Objectives

- บรรยายความหมาย ความสำคัญ และโครงสร้างของกระบวนการ (processes)
- อธิบายหน้าที่ การทำงานของระบบปฏิบัติการในส่วนการจัดการกระบวนการ
- เข้าใจและสามารถเขียนโปรแกรมเพื่อการสื่อสารระหว่างกระบวนการได้
- บรรยายความหมาย ประโยชน์ และการทำงานของเธร็ด (threads)
- แนะนำคลังคำสั่งในการเขียนโปรแกรมแบบหลายเธร็ด
- แนะนำแนวทางการจัดการกระบวนการและการจัดการเธร็ดในระบบปฏิบัติการในปัจจุบัน

Contents

- What is a process?
- Inside a process
- Management of processes
- Inter process communications (IPC)
- A brief view of processes in Unix and Android

- What is a thread and its benefits?
- Management of threads
- Thread Libraries
- Programming of server-client communication

Credits

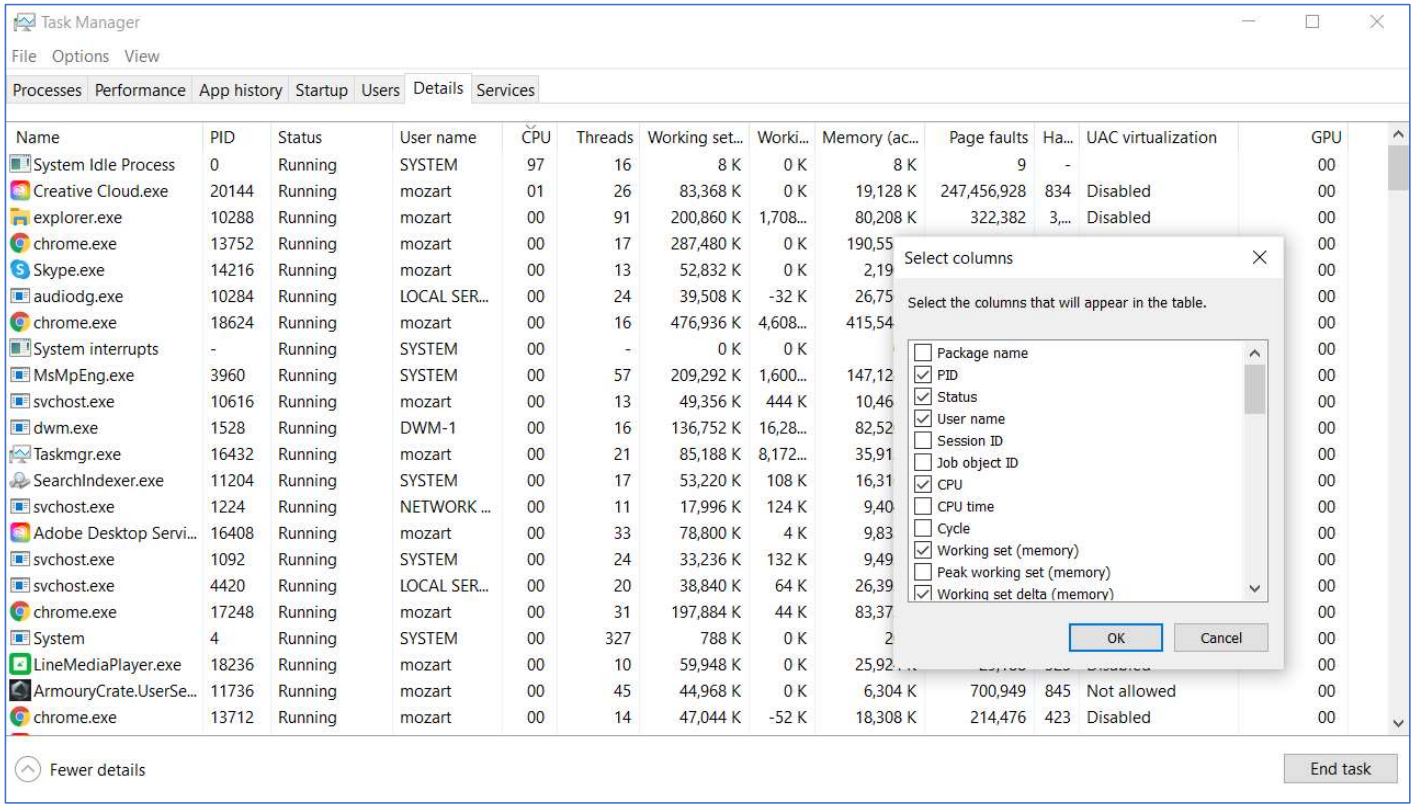
Many pictures in this slides from obtained from Google image search and might be copyright protected by their creators.

What is a Process?

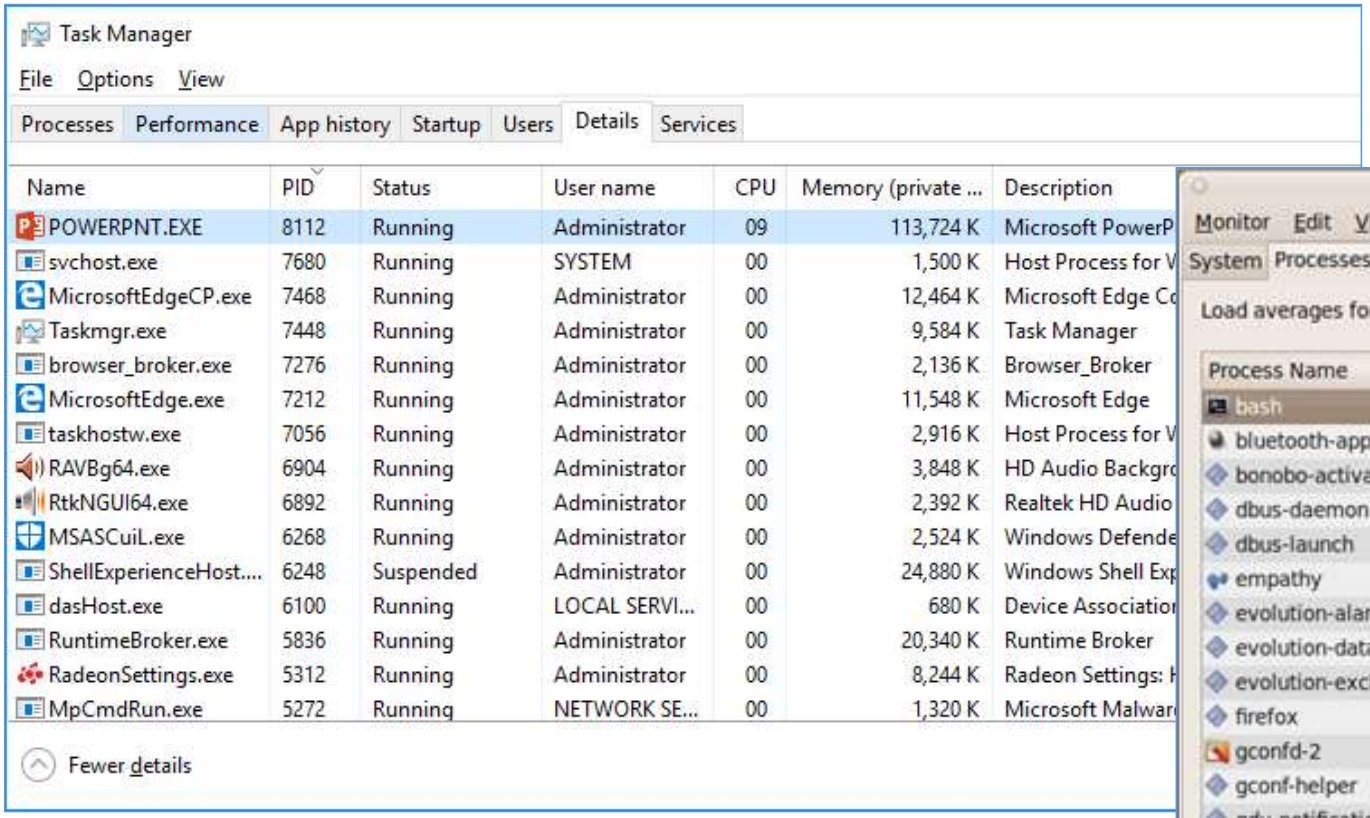
- A program in execution with necessary related information.
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system instructions
- A living form of a program. Very similar to “job” or “task”.

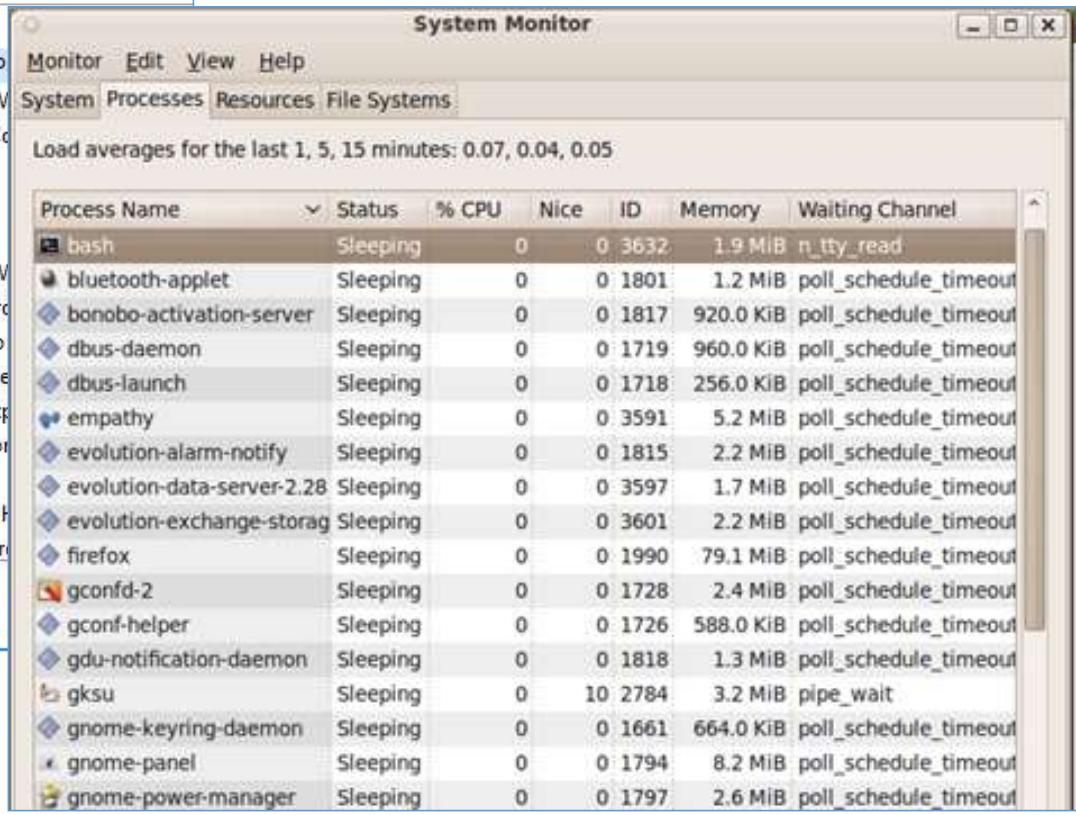
Process Elements

- Identifier
- State
- Priority
- Program counter
- Memory pointers
- Context data
- I/O status information
- Accounting information



Process status in Windows and Ubuntu Linux





Process status *ps* and *top* commands

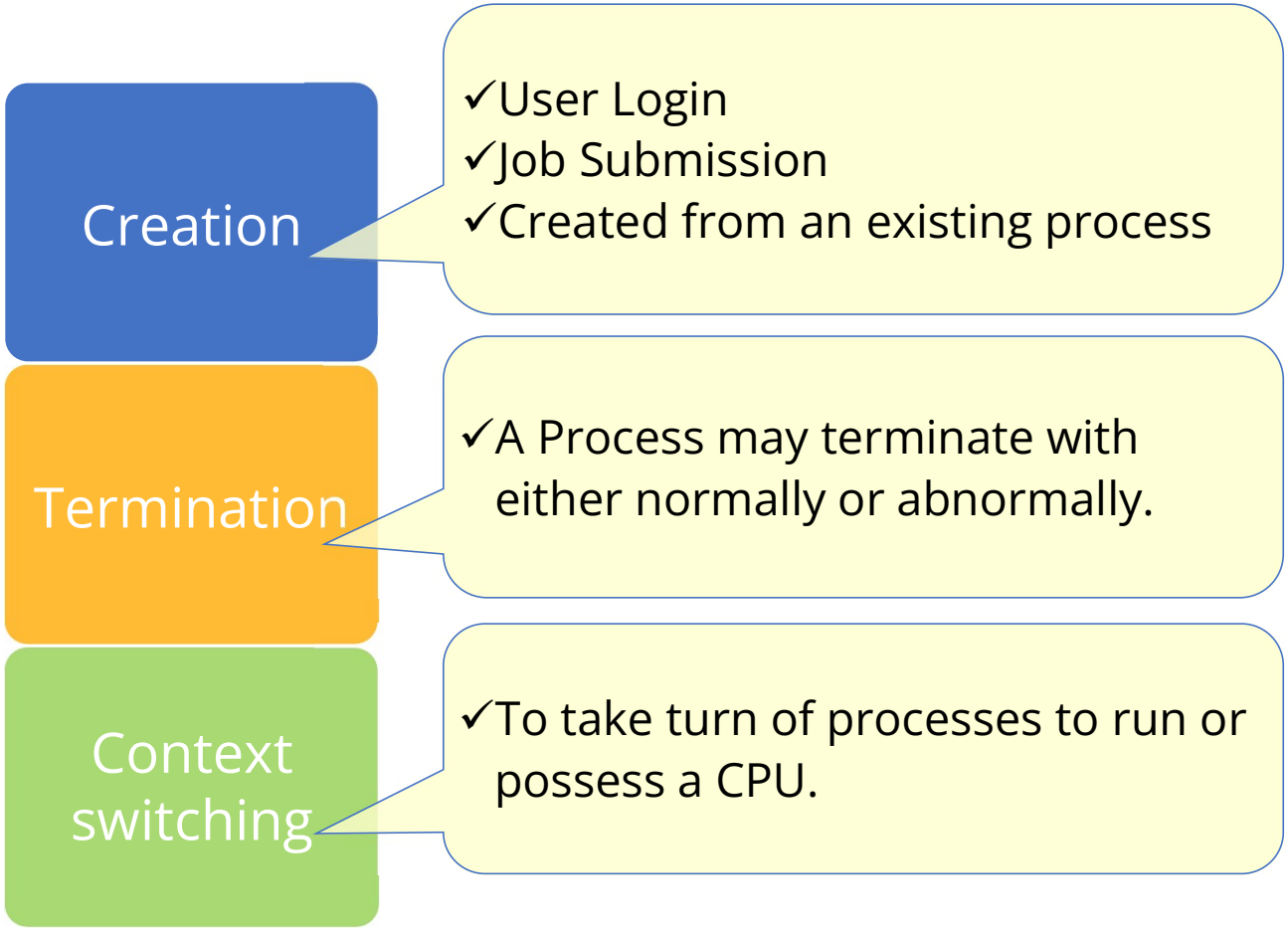
ps -ux

```
File  Edit  View  Terminal  Help
mozart@mozartubun:~$ ps -ux
Warning: bad ps syntax, perhaps a bogus '-'? See http://pr
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START
mozart    1661  0.0  0.1  40700  3564 ?        Sl   12:11
mozart    1676  0.0  0.1  25540  6512 ?        Ssl  12:11
mozart    1715  0.0  0.0   4916   612 ?        Ss   12:11
mozart    1718  0.0  0.0   3380   704 ?        S    12:11
mozart    1719  0.0  0.0   3296  1796 ?        Ss   12:11
mozart    1723  0.0  0.1  94140  4676 ?        Ssl  12:11
mozart    1726  0.0  0.0  10628  2908 ?        S    12:11
mozart    1728  0.0  0.1   7592  4628 ?        S    12:11
mozart    1737  0.0  0.2  96452  8780 ?        Ssl  12:11
```

top command

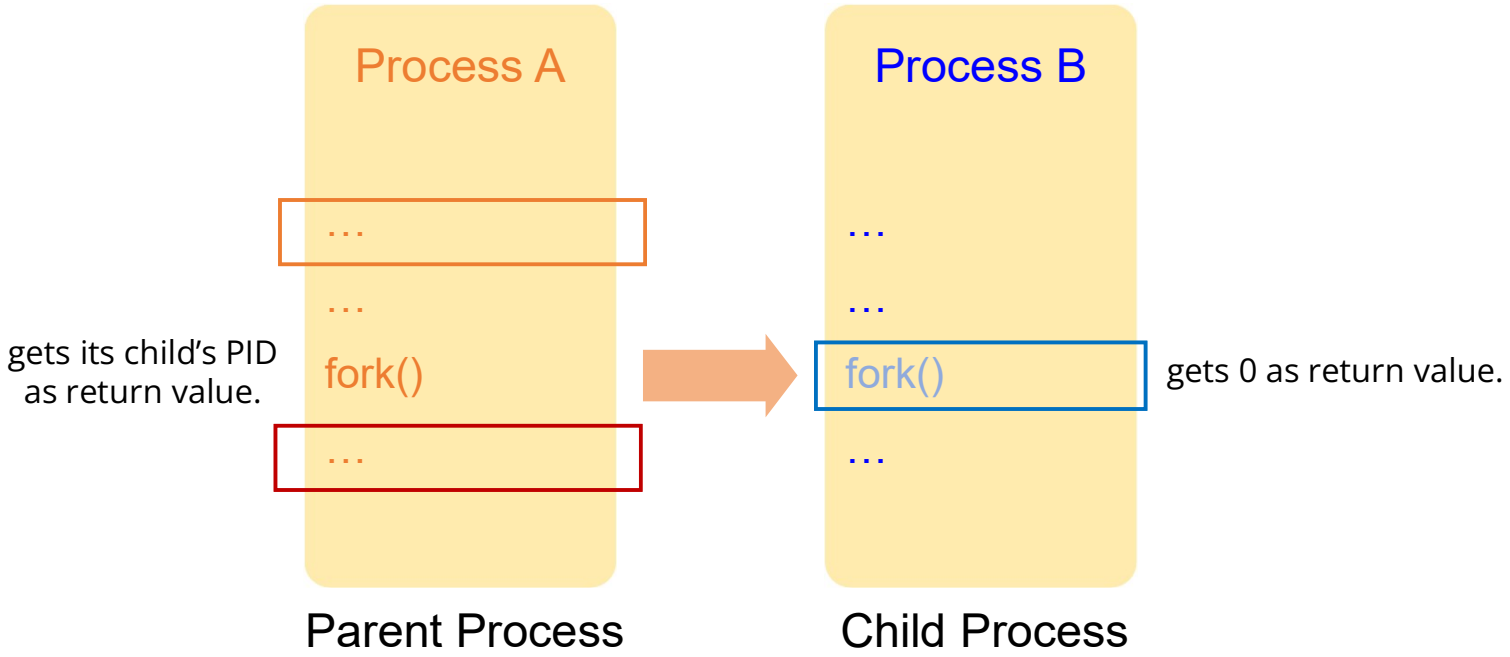
```
tecmint@TecMint ~
top - 11:58:28 up  1:31,  1 user,  load average: 0.41, 0.73, 0.77
Tasks: 235 total,   1 running, 234 sleeping,   0 stopped,   0 zombie
%Cpu(s):  8.5 us,  1.8 sy,  0.4 ni, 82.7 id,  6.7 wa,  0.0 hi,  0.0 si,
Kib Mem :  8069036 total,   564660 free,  4203844 used,  3300532 buff/c
Kib Swap:  3906556 total,  3900432 free,     6124 used.  3001872 avail

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COM
 5411 tecmint   20   0 2569304 642700 154820 S  12.5  8.0   6:15.54 Web
 8213 tecmint   20   0 1084812 304592 123508 S   6.2  3.8   0:20.64 chr
    1 root      20   0  185656   6192   3912 S   0.0  0.1   0:01.82 sys
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kth
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.07 kso
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kwo
    7 root      20   0       0      0      0 S   0.0  0.0   0:08.86 rcu
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu
    9 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 mig
   10 root      rt   0       0      0      0 S   0.0  0.0   0:00.02 wat
   11 root      rt   0       0      0      0 S   0.0  0.0   0:00.02 wat
   12 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 mig
   13 root      20   0       0      0      0 S   0.0  0.0   0:00.10 kso
   15 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kwo
   16 root      rt   0       0      0      0 S   0.0  0.0   0:00.02 wat
top.txt
```

Management of Processes



Process Creation

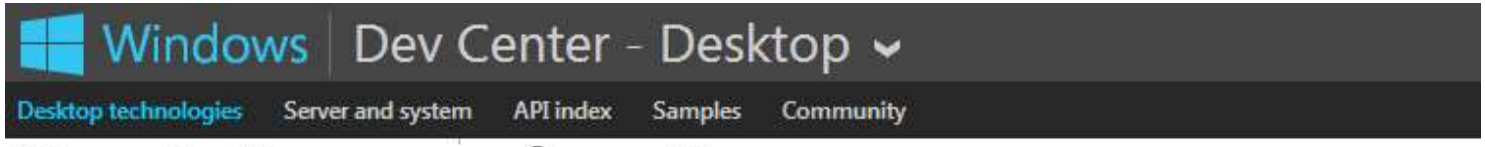
- A process may create another process with *fork()* system call in UNIX-based OS.



fork() system call

```
1.#include <stdio.h>
2.#include <unistd.h>
3.int main(int argc, char *argv[])
4.{
5.      int pid;
6.      /* create another process */
7.      pid = fork();
8.      if (pid < 0) { /* error occurred */
9.              fprintf(stderr, "Fork Failed");
10.             exit(-1);
11.     }
12.     else if (pid == 0) { /* child process */
13.             execlp("./child","child",NULL);
14.     }
15.     else { /* parent process waits for */
16.            /* the child to complete */
17.             wait(NULL);
18.             printf("Child completed");
19.             exit(0);
20.     }
```

Windows's CreateProcess()



CreateProcess function

Creates a new process and its primary thread. The new process runs in the security context of the calling process.

If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the CreateProcessAsUser or CreateProcessWithLogonW function.

Syntax

C++

```
BOOL WINAPI CreateProcess(
  _In_opt_     LPCTSTR lpApplicationName,
  _Inout_opt_  LPTSTR lpCommandLine,
  _In_opt_     LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_     LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_         BOOL bInheritHandles,
  _In_         DWORD dwCreationFlags,
  _In_opt_     LPVOID lpEnvironment,
  _In_opt_     LPCTSTR lpCurrentDirectory,
  _In_         LPSTARTUPINFO lpStartupInfo,
  _Out_        LPPROCESS_INFORMATION lpProcessInformation
);
```

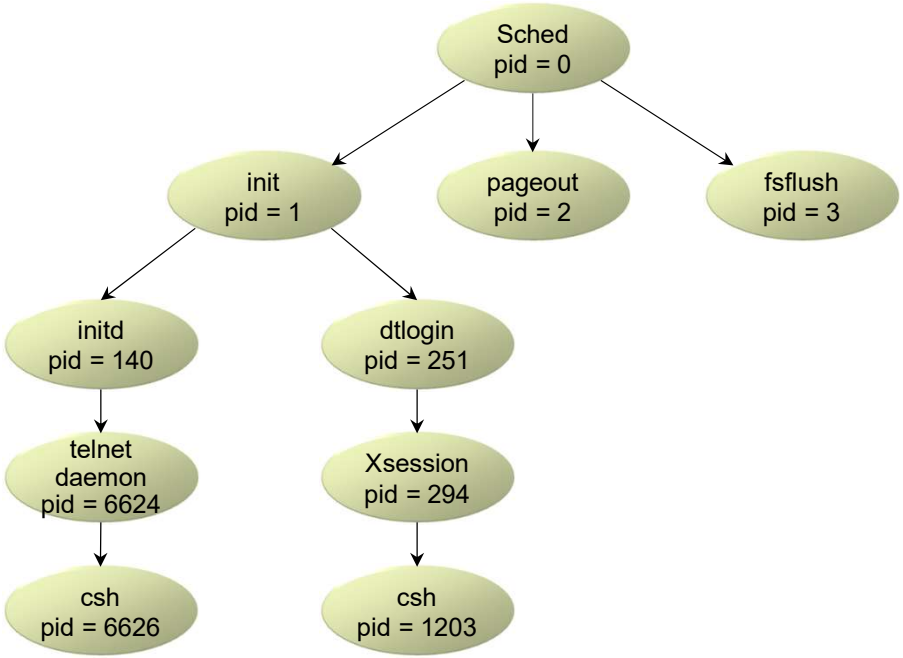
Windows's CreateProcess()

```
1.#include <windows.h>
2.#include <stdio.h>
3.void main(VOID)
4.{
5.     STARTUPINFO si;
6.     PROCESS_INFORMATION pi;
7.     ZeroMemory( &si, sizeof(si) );
8.     si.cb = sizeof(si);
9.     ZeroMemory( &pi, sizeof(pi) );
10.     // Start the child process.
11.     if( !CreateProcess( NULL,   // No module name (use command line).
12.         TEXT("MyChildProcess"), // Command line.
13.         NULL,             // Process handle not inheritable.
14.         NULL,             // Thread handle not inheritable.
15.         FALSE,            // Set handle inheritance to FALSE.
16.         0,                // No creation flags.
17.         NULL,             // Use parent's environment block.
18.         NULL,             // Use parent's starting directory.
19.         &si,              // Pointer to STARTUPINFO structure.
20.         &pi )             // Pointer to PROCESS_INFORMATION structure.
21.     ) {
22.         printf( "CreateProcess failed (%d).\n", GetLastError() );
23.         return;
```

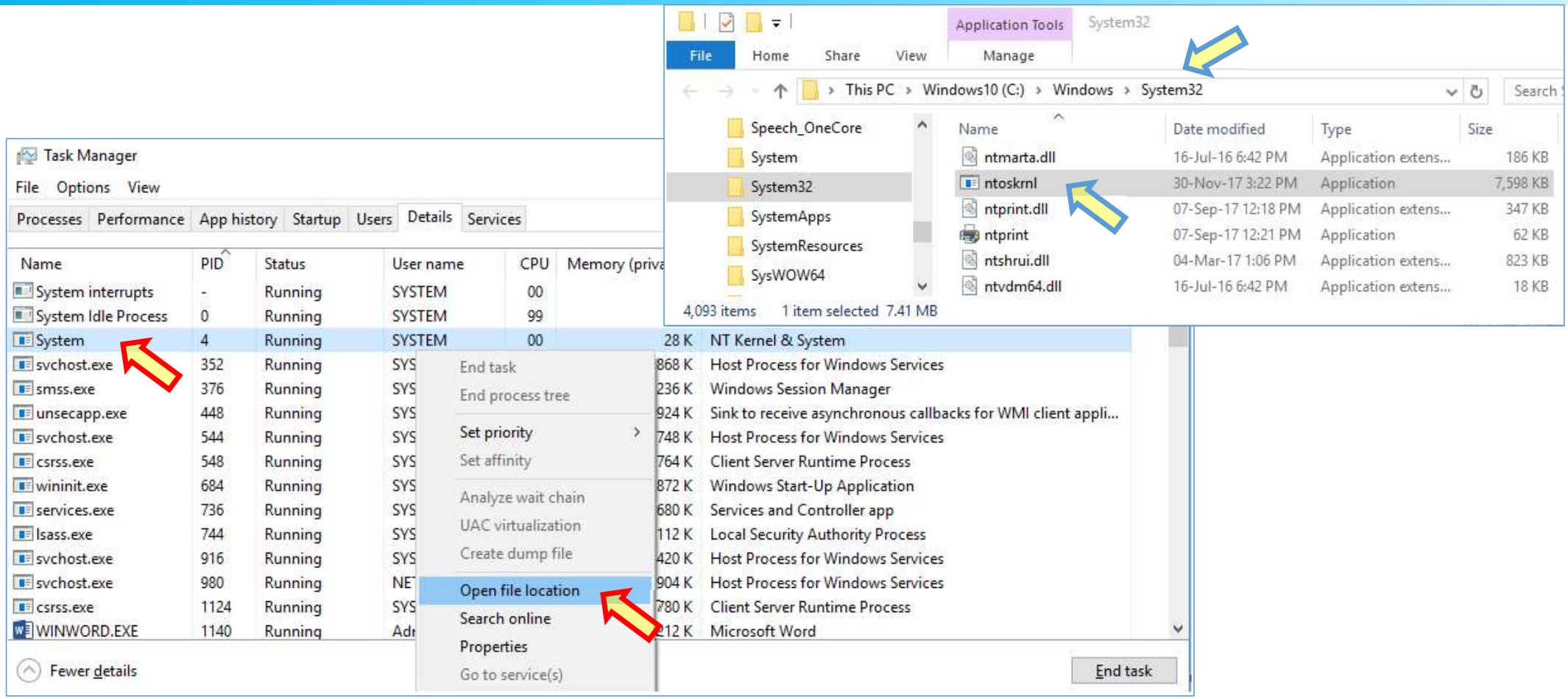
```
25.      // Wait until child process exits.
26.     WaitForSingleObject( pi.hProcess, INFINITE );
27.     // Close process and thread handles.
28.     CloseHandle( pi.hProcess );
29.     CloseHandle( pi.hThread );
30.}
```

Process Hierarchy in Unix

- The first 3 processes in a typical Unix are init, pageout, and fsflush.
- The process ***init*** creates all other processes in Unix.



System Processes in Windows 10



Process Termination

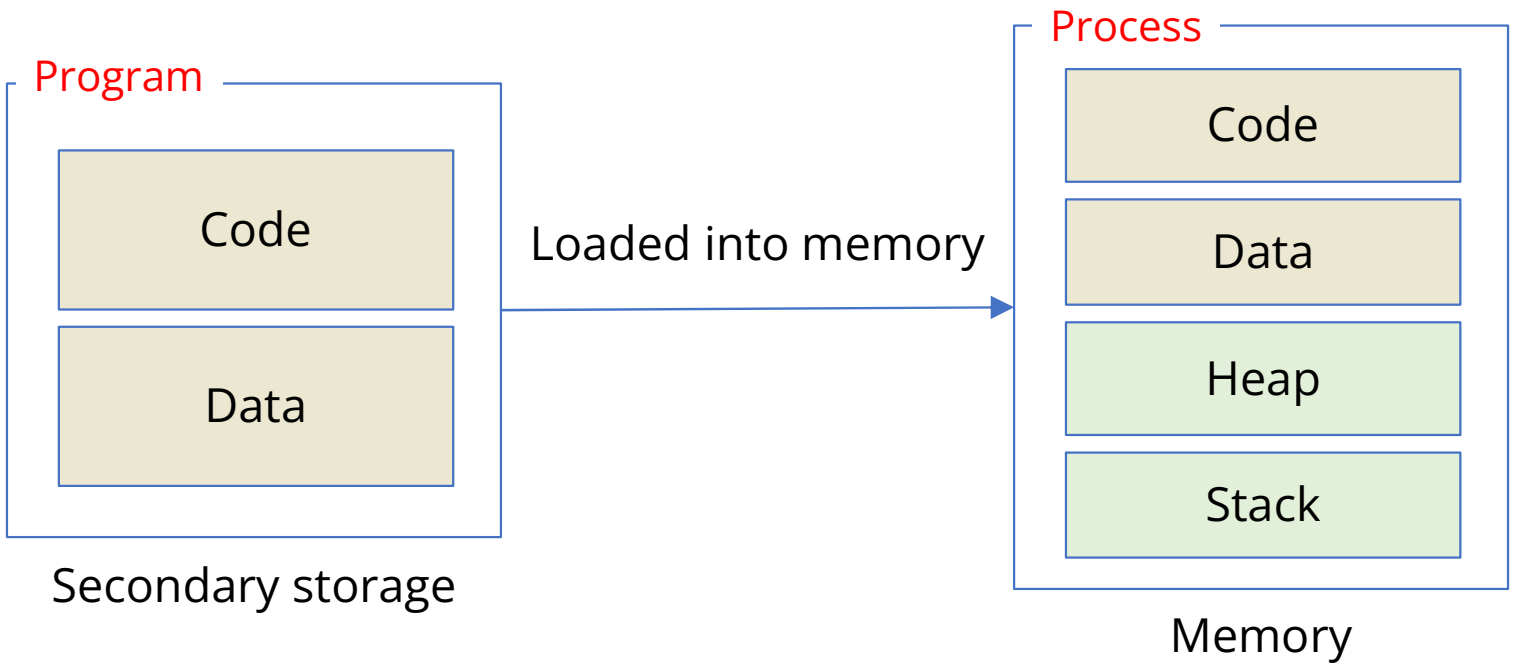
- Causes of process termination:
  - Normal completion: process ends normally with *exit()* system call.
  - Process exceeds its time quota.
  - Process requires more memory than the system limit.
  - Process violates others' memory or other resources.
  - Faults in I/O devices operation
  - Invalid or unprivileged instruction
  - Arithmetic error: e.g. division by 0.
  - When its parent process terminates, the OS may automatically terminate its child processes.

Contents

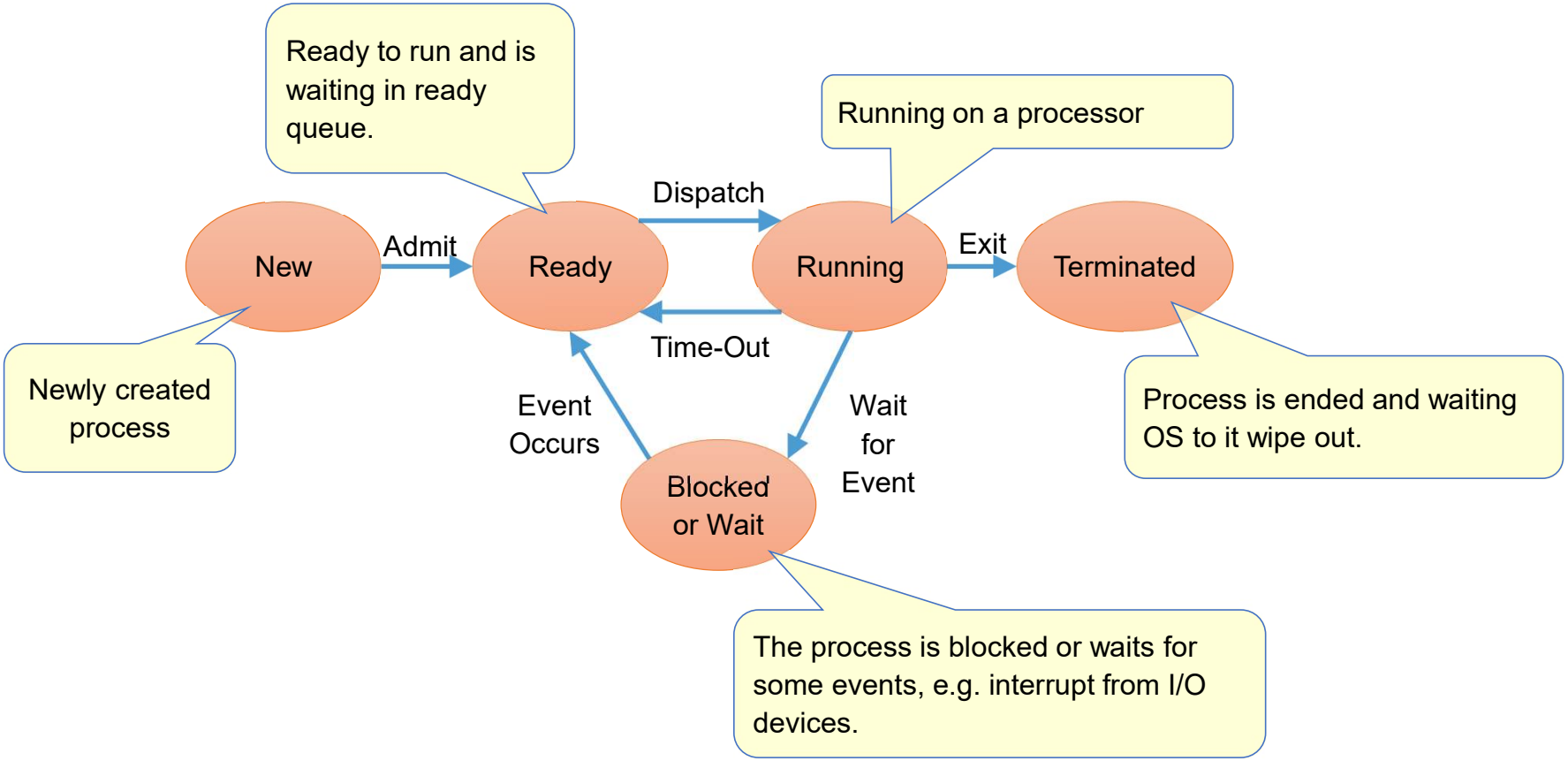
- What is a process?
- Inside a process
- Management of processes
- Inter process communications (IPC)
- A brief view of processes in Unix and Android

- What is a thread and its benefits?
- Management of threads
- Thread Libraries
- Programming of server-client communication

What are inside a Process?

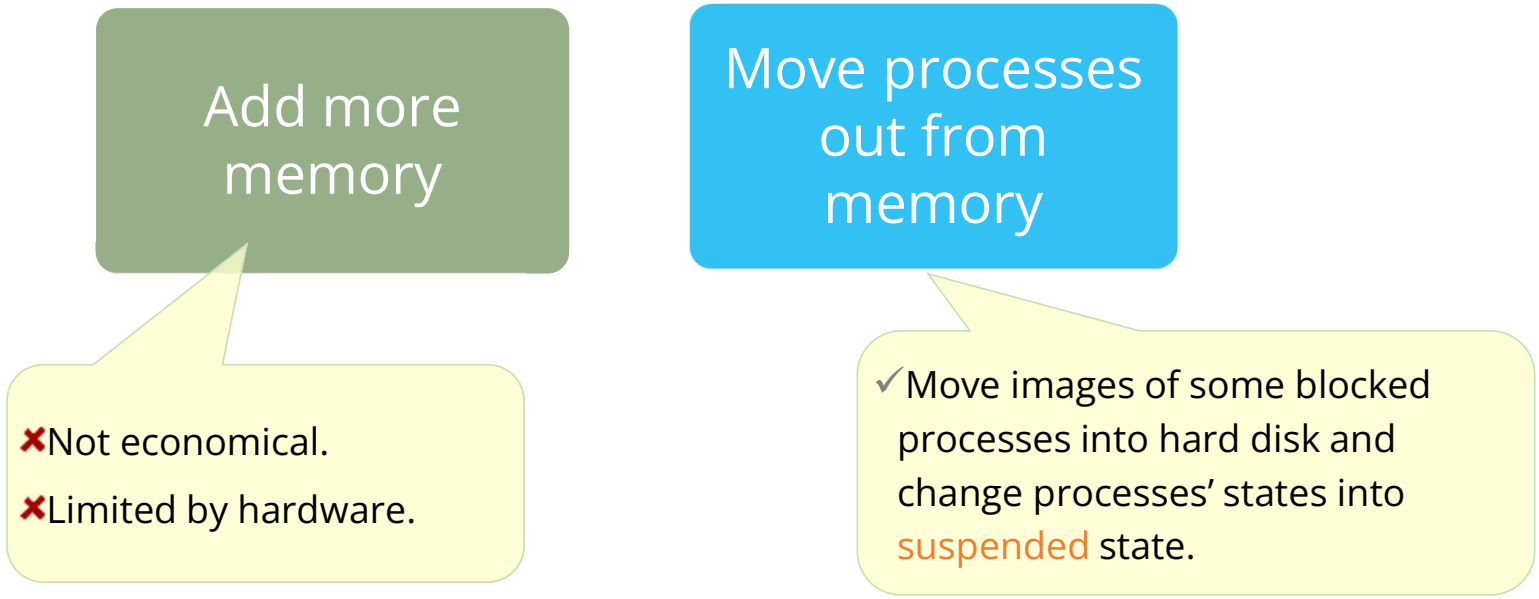


Process States & Life Cycle

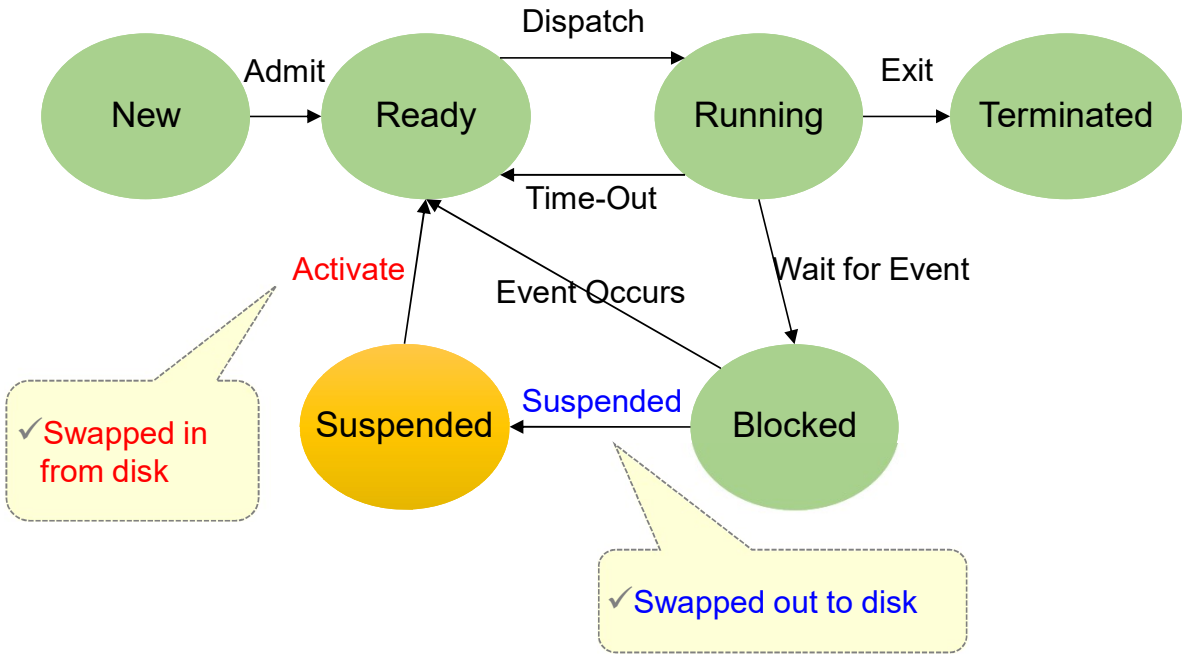


Suspended Processes

- When most or all processes are waiting for events to occur, no process is in ready state or running, resulting in low processor utilization.
- Solutions?

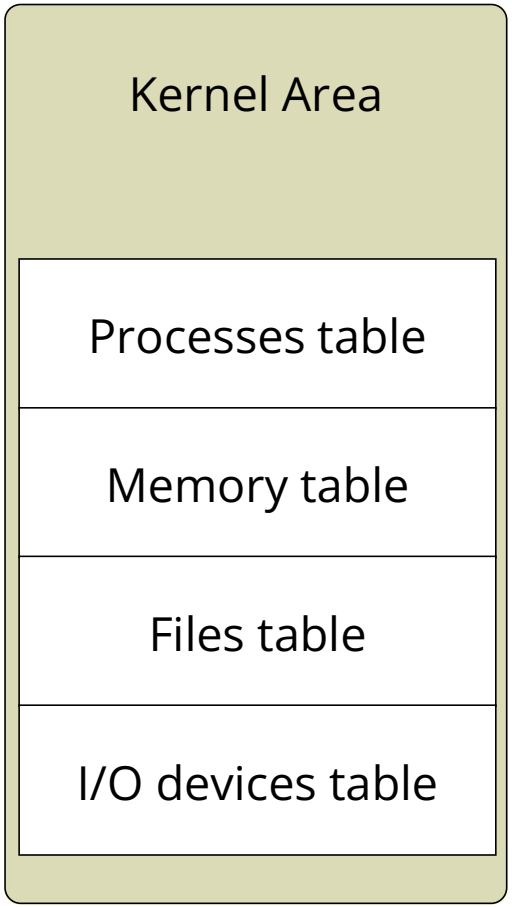


Process Life Cycle with Suspended State



Essential OS Control Structures

- A process may use several types of computer resources:
  - Processor
  - Memory
  - Files
  - I/O devices.
- Therefore, it has a lot of information associated with to support throughout its life cycle.
- These data structures are very important and, so, are maintained in kernel.



Operating System Control Structures

- Information about the current status of each process and resource
- Tables are constructed for each entity the operating system manages

Memory Table

- Allocation of main memory to processes
- Protection attributes for access to shared memory regions
- Information needed to manage virtual memory

I/O Table

- I/O device is available or assigned
- Status of I/O operation
- Location in main memory being used as the source or destination of the I/O transfer

Operating System Control Structures (cont.)

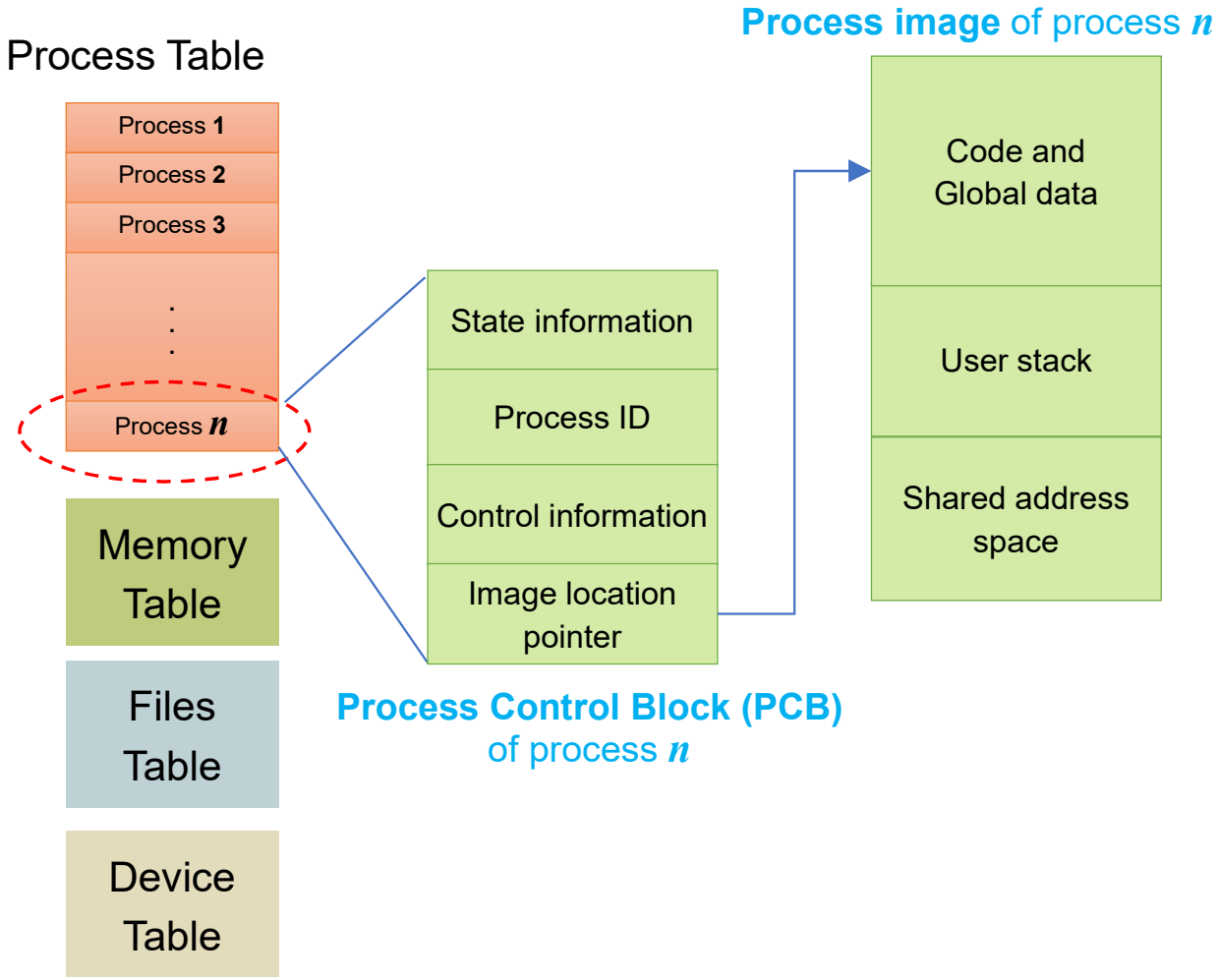
File Table

- Existence of files
- Location on secondary memory
- Current Status
- Attributes
- Sometimes this information is maintained by a file management system

Process Table

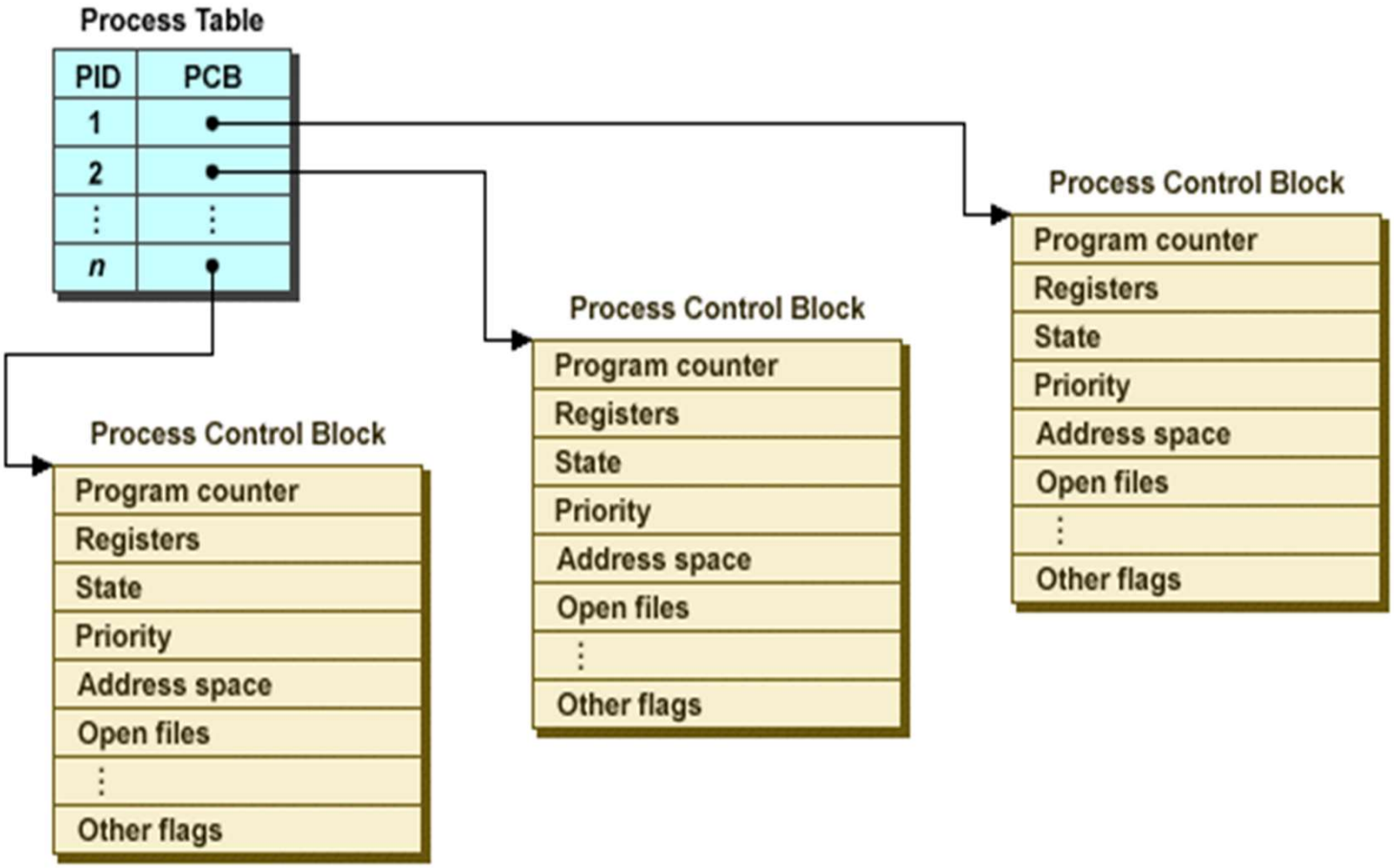
- Array or list of *Process Control Block* (PCB)
- Contains the process elements
- Created and manage by the operating system
- Allows support for multiple processes

Process table and PCB



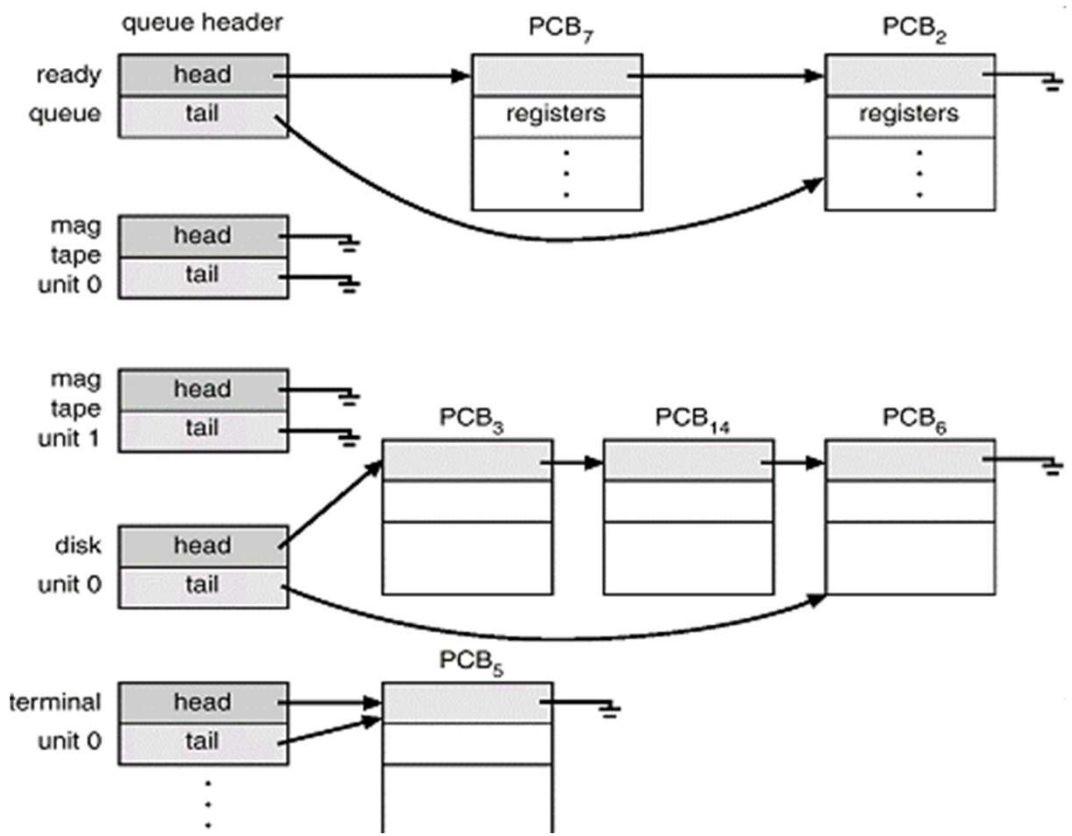
- Components of a PCB
  - ID, Identifier
  - State
  - Priority
  - Program counter
  - Memory pointers
  - Context data (repository of user-visible registers)
  - I/O status information
  - Accounting information

An example of process table and PCB's



PCB Lists

- OS maintains several linked-lists for running the system and managing all resources such as disks, ready queue, etc.



Contents

- What is a process?
- Inside a process
- Management of processes
- Inter process communications (IPC)
- A brief view of processes in Unix and Android

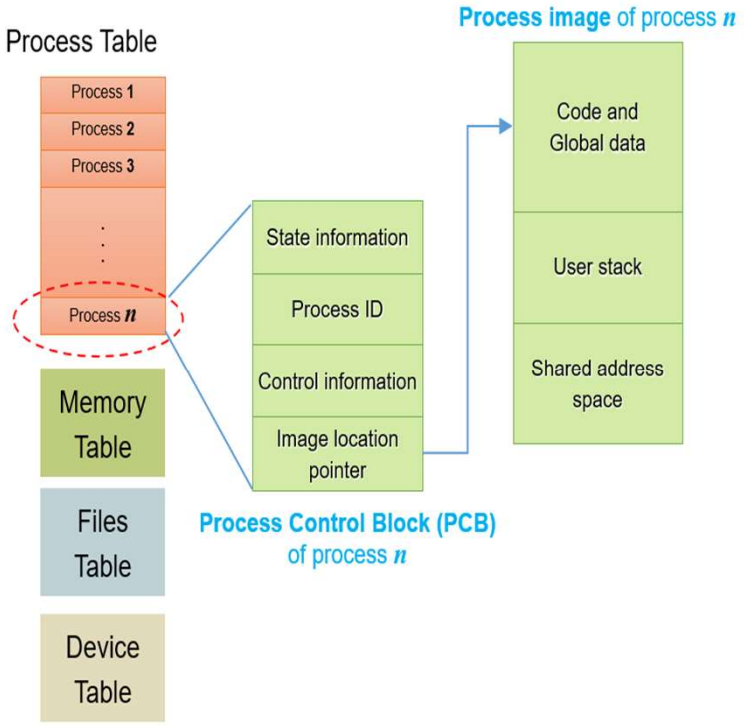
- What is a thread and its benefits?
- Management of threads
- Thread Libraries
- Programming of server-client communication

OS's tasks for managing of processes

- Creating a Process
- Changing Mode
- Process/Context Switching

OS's tasks for creating a process

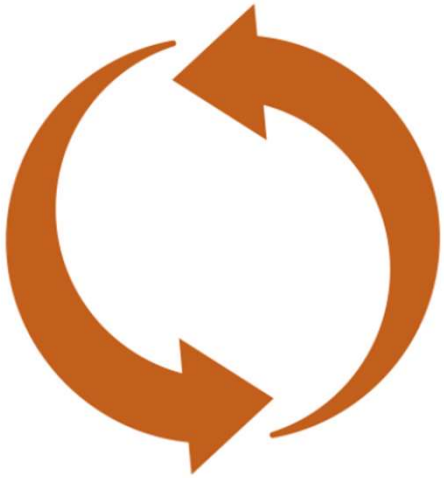
1. Assign a new process ID
2. Allocate process image area
3. Prepare PCB
4. Link control pointers
5. Other tasks, e.g. accounting usage



- Process Control Table (PCB)
  - Contains the process elements
  - Created and manage by the operating system
  - Allows support for multiple processes
- Components of a PCB
  - ID, Identifier
  - State
  - Priority
  - Program counter
  - Memory pointers
  - Context data
  - I/O status information
  - Accounting information

OS's tasks for changing mode

- Recall that CPU (or machine) runs in (at least) 2 modes:
  1. *Kernel mode*, hardware can be totally controlled by OS.
  2. *User mode*, with limited use for user processes
- When to switch to kernel mode?
  - an exception such as an *interrupt*, a *fault*, or a trapping *system call*.
- How?
  - Changing this mode with bit(s) in the Program Status Word (PSW) register (EFLAGS register in Intel-based).



Intel 64/32-bit kernel modes

- Current Privilege Level (CPL) is defined as the protection level of the currently executing code segment.
- Bits 0 and 1 of the CS segment register.
  - CPL < 3 = kernel modes
  - CPL = 3 → user mode

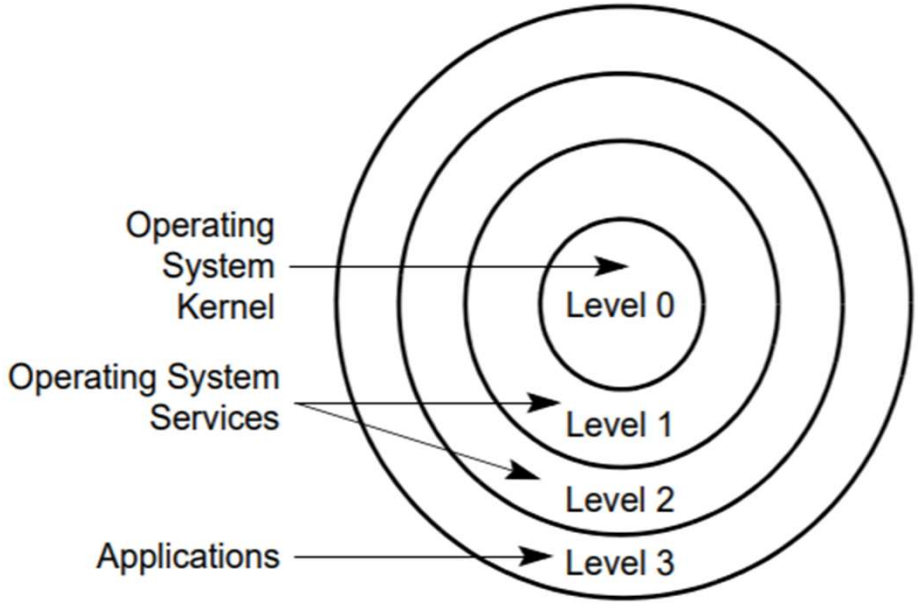


Figure 5-3: Protection rings
(Intel Architecture 64 and 32-bit Manual)

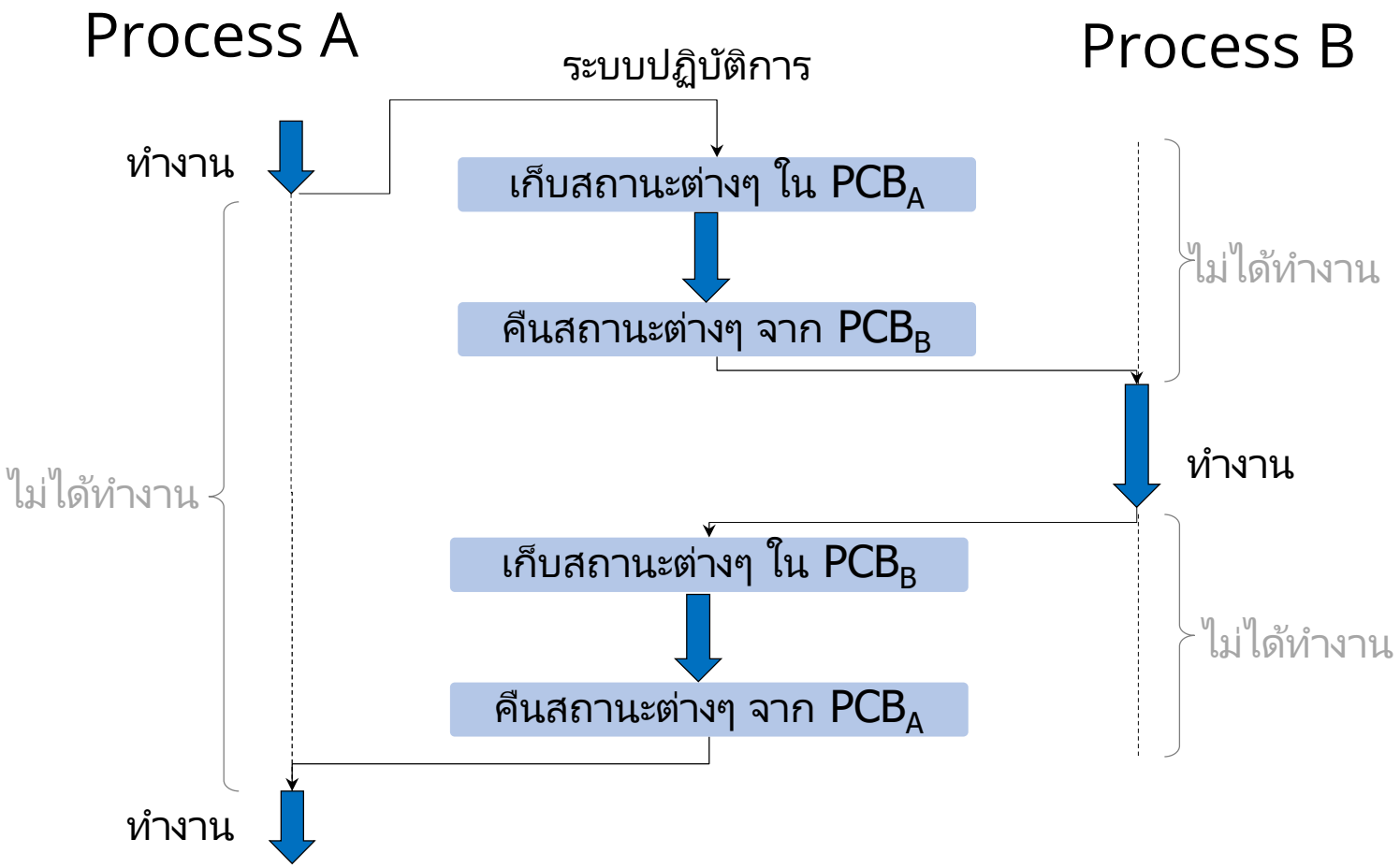
Context Switching

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.
  - This context saving and loading is called ***context switching*** or ***process switching***.
  - performed by ***dispatcher*** module of OS.
  - is overhead; the system does no useful work while switching. Time dependent on hardware support.
- Context switching occurs from ...
  - Interrupts
  - Traps
  - System calls

Steps in Context Switching

1. Save process context in PCB
2. Change states information (user-visible registers)
3. Select next running process
4. Rearrange PCB links
5. Update the to-be-running process
6. Set PC to run the next process

Switching processes A and B



Contents

- What is a process?
- Inside a process
- Management of processes
- Inter process communications (IPC)
- A brief view of processes in Unix and Android

- What is a thread and its benefits?
- Management of threads
- Thread Libraries
- Programming of server-client communication

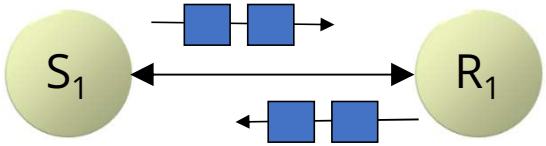
Inter-Process Communication

- Fundamental IPC mechanism is message passing.
- System calls: *send()* and *receive()*

- *Basic principles* in design consideration of message passing:
  - Direct vs. indirect
  - Synchronization

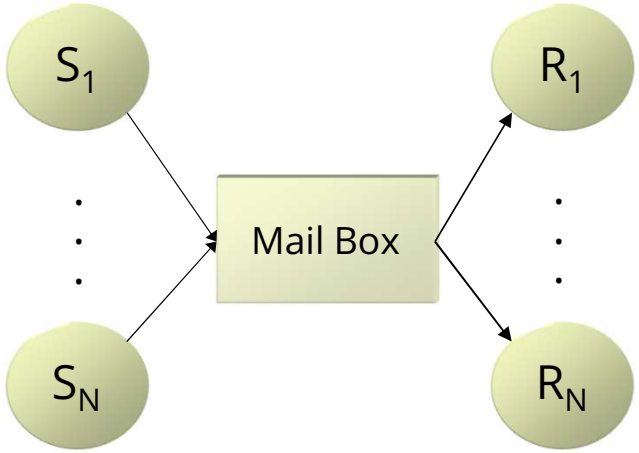
Direct vs. Indirect Message Passing

- Direct communication
  - Clearly indicates its partner's PID
  - Participants must know each other's PID.



Direct Communication

- Indirect communication
  - Uses mailbox, or port, not direct PID.
  - The mailbox can be shared among more than 2 processes.



Indirect Communication

Synchronization Mode

1. Blocking (or Synchronous) mode.
   - The sender waits until the receiver has processed the message before continuing.
2. Non-blocking (or Asynchronous) mode.
   - The sender continues to execute code without waiting for the receiver to process the message.

Various methods for Inter-Process Communication (IPC)

- Pipes
- FIFOs
- Sockets
  - Stream vs Datagram (vs Seq. packet)
  - UNIX vs Internet domain
- POSIX message queues
- POSIX shared memory
- POSIX semaphores
- System V message queues
- System V shared memory
- System V semaphores

Communication

- Shared memory mappings
  - File vs anonymous
- Cross-memory attach
  - proc_vm_readv() / proc_vm_writev()
- Signals
  - Standard, Realtime

Signal

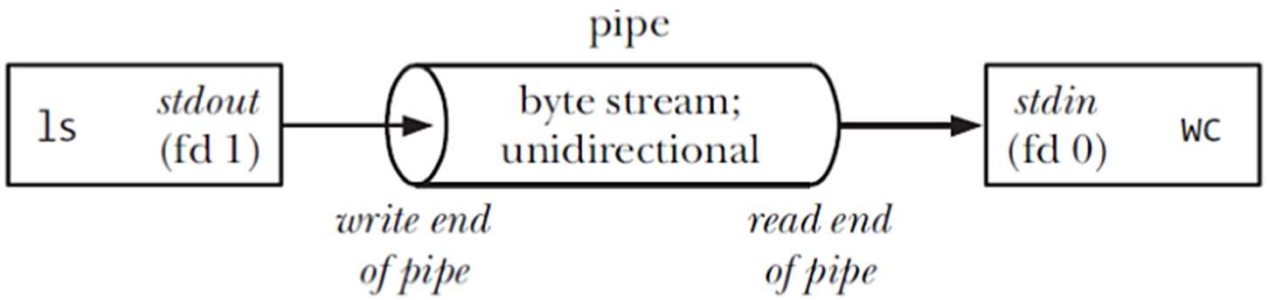
- Eventfd
- Futexes
- Record locks
- File locks
- Mutexes
- Condition variables
- Barriers
- Read-write lock

Synchronization

Pipe

- There are 2 types of pipe in Unix/Linux:
  1. Unnamed pipe
  2. Named pipe

Example of unnamed pipe → `ls | wc -l`



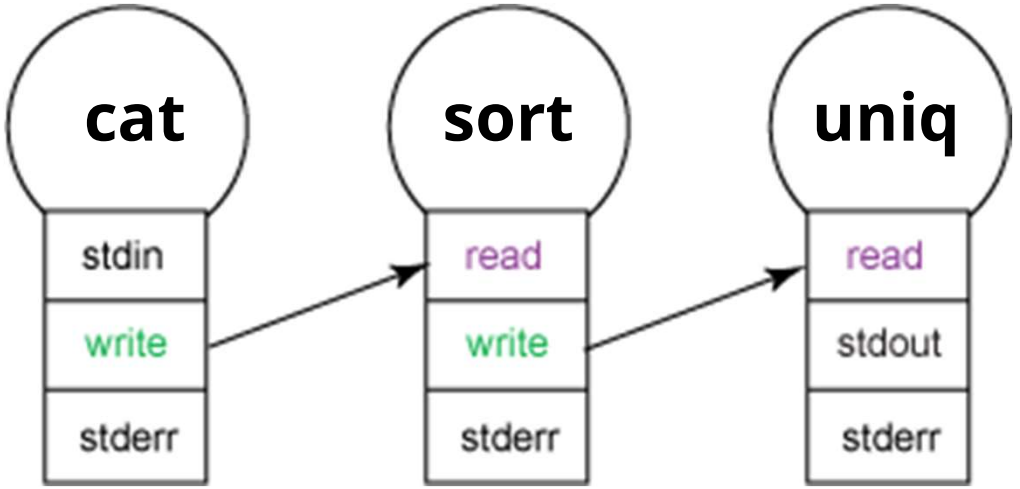
A common usage of *unnamed* pipe

Unnamed Pipe

```
% cat test.dat | sort | uniq
```

```
this
is
the
way
the
world
ends
```

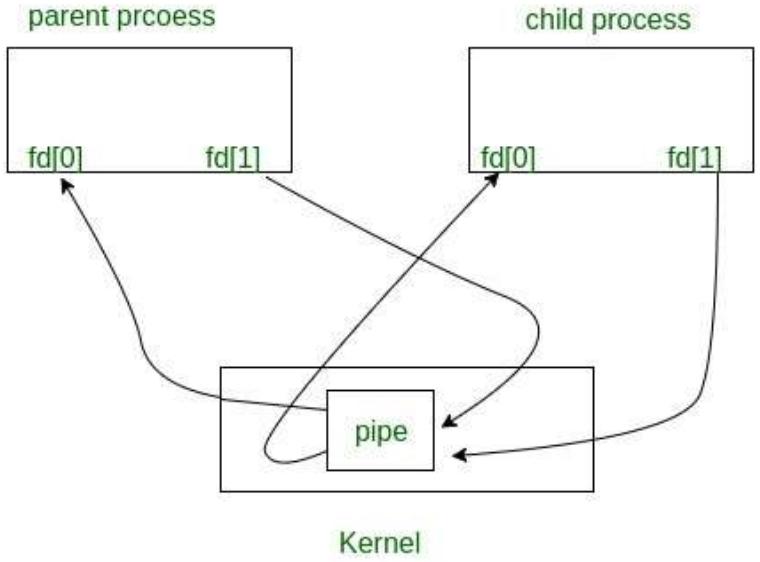
```
ends
is
the
this
way
world
```



Source: https://opensource.com/article/19/4/interprocess-communication-linux-channels

Example of programming for unnamed pipe

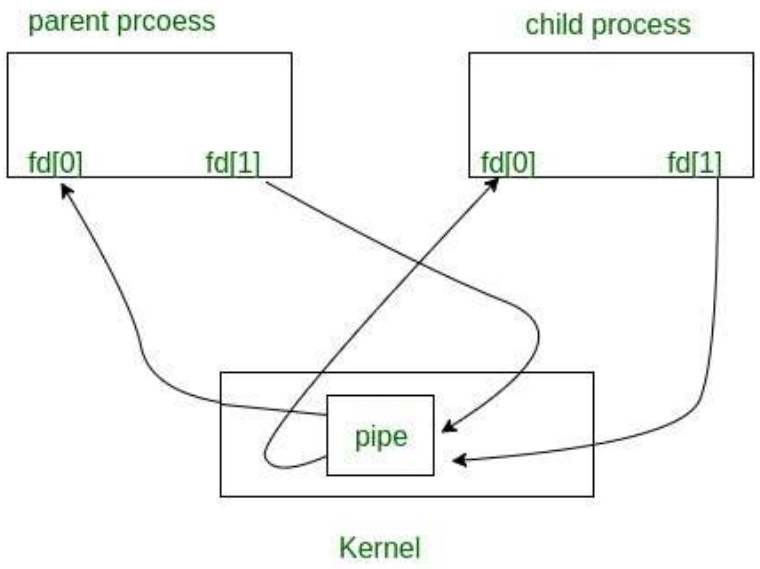
```c
#include <unistd.h>   /* read, write, pipe, _exit */
#include <string.h>

#define ReadEnd  0
#define WriteEnd 1

void report_and_exit(const char* msg) {
  perror(msg);
  exit(-1);    /** failure **/
}

int main() {
  int pipeFDs[2]; /* two file descriptors */
  char buf;       /* 1-byte buffer */
  const char* msg = "Nature's first green is gold\n"; /* bytes to write */

  if (pipe(pipeFDs) < 0) report_and_exit("pipeFD");
  pid_t cpid = fork();                              /* fork a child process */
  if (cpid < 0) report_and_exit("fork");            /* check for failure */
```



Source: https://opensource.com/article/19/4/interprocess-communication-linux-channels

Example of programming for unnamed pipe (cont.)

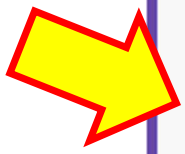
```
  if (0 == cpid) {    /*** child ***/                  /* child process */
    close(pipeFDs[WriteEnd]);                          /* child reads, doesn't write

    while (read(pipeFDs[ReadEnd], &buf, 1) > 0)        /* read until end of byte stre
      write(STDOUT_FILENO, &buf, sizeof(buf));         /* echo to the standard output

    close(pipeFDs[ReadEnd]);                           /* close the ReadEnd: all done
    _exit(0);                                          /* exit and notify parent at o
  }
  else {              /*** parent ***/
    close(pipeFDs[ReadEnd]);                           /* parent writes, doesn't read

    write(pipeFDs[WriteEnd], msg, strlen(msg));        /* write the bytes to the pipe
    close(pipeFDs[WriteEnd]);                          /* done writing: generate eof *

    wait(NULL);                                        /* wait for child to exit */
    exit(0);                                           /* exit normally */
  }
  return 0;
}
```



Source: https://opensource.com/article/19/4/interprocess-communication-linux-channels

Named pipe

- A *FIFO*, or *named pipe*, is a special file similar to a pipe but with a name on the filesystem. Multiple processes can access this special file for reading and writing like any ordinary file.
- Thus, *the name works only as a reference point for processes that need to use a name in the filesystem*.
- A FIFO has the same characteristics as any other file. For example, it has ownership, permissions, and metadata. *Its file type is '**p**'.*



```
$ mkfifo pipe1
$ mknod pipe2 p
$ ls -l
prw-r--r-- 1 cuau cuau 0 Oct 7 21:17 pipe1
prw-r--r-- 1 cuau cuau 0 Oct 7 21:17 pipe2
```

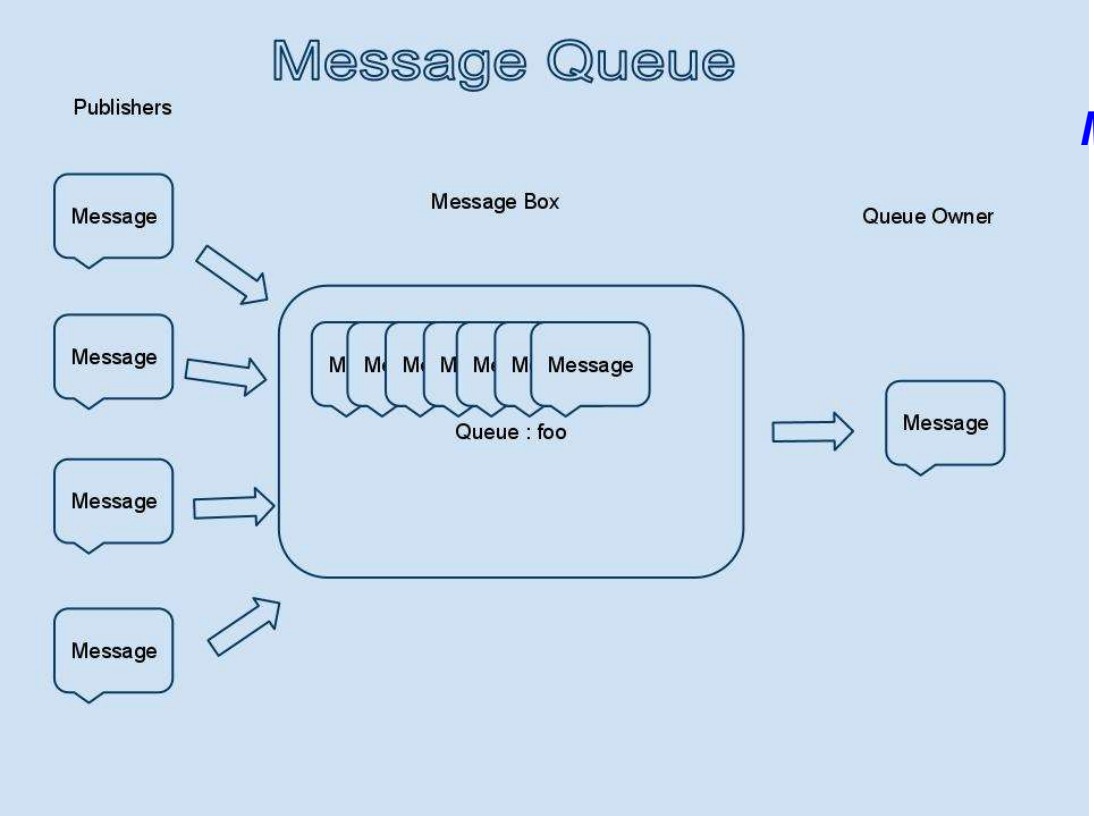
Unnamed pipe or Named pipe?

Further self-study:

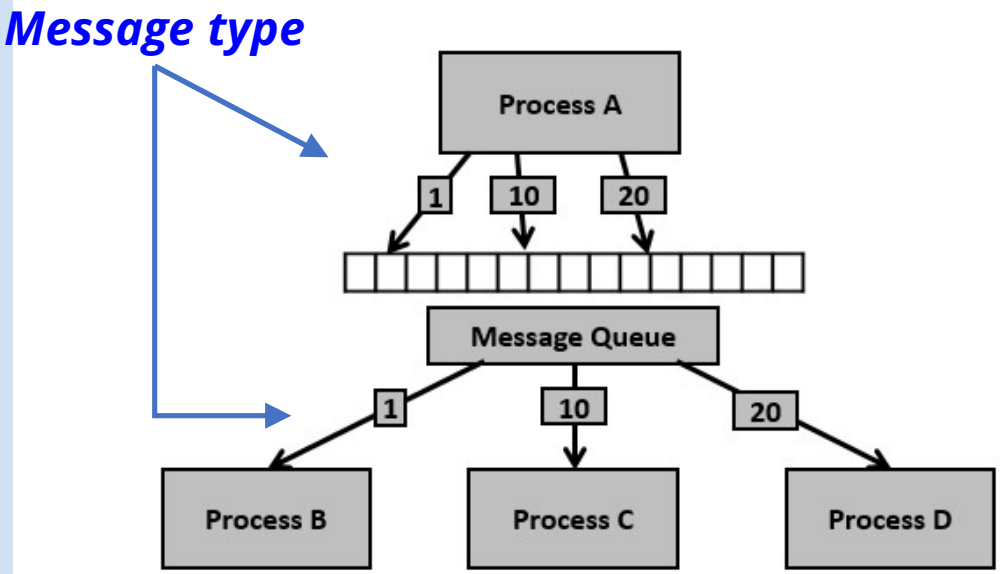
- What are the main advantages/disadvantages of named pipe over unnamed pipe?
- Where exactly is the named pipe? Is it on the disk or what?

More about pipe() at https://linux.die.net/man/7/pipe

System V's MQ (Message Queue)

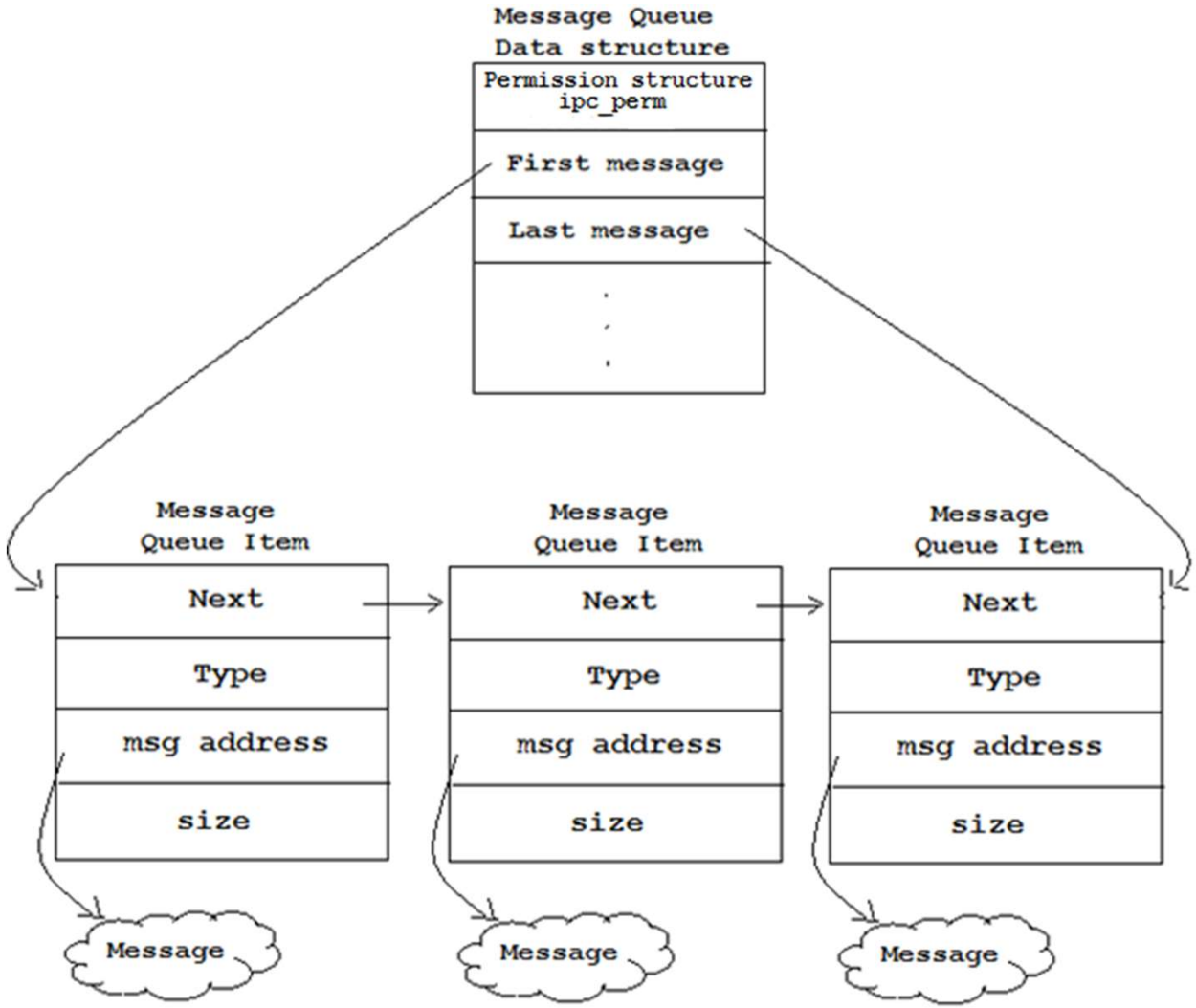


Message Passing Architecture

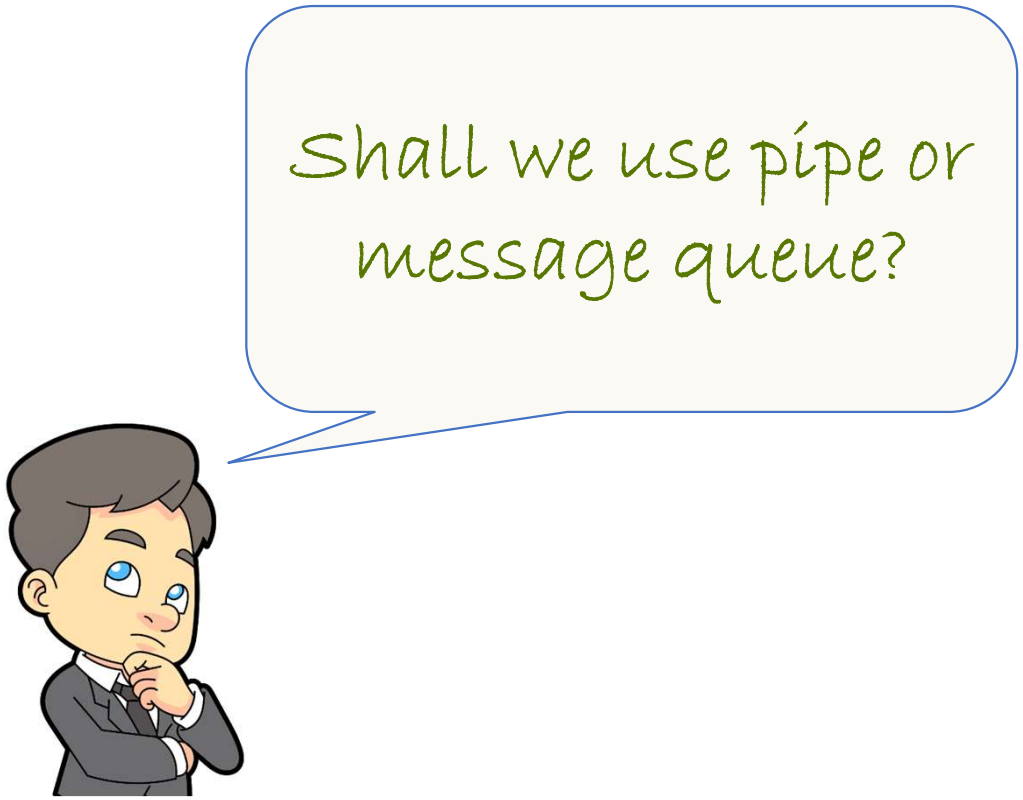


More at: https://www.tutorialspoint.com/ipc-using-message-queues

How a MQ is handled in a Linux kernel?



Thinking



Pipe vs Message Q?

Source: https://www.tutorialspoint.com/difference-between-pipes-and-message-queues

Key	Pipe	Message Queue
Concept	Unix IPC form to provide a flow of information in one direction.	System V IPC form to store a list of messages.
Creation	Created using pipe() function which returns two file descriptors, one is for reading and another is for writing.	Created using msgget() function which returns a queue identifier.
Direction	The pipe is *unidirectional*.	A message queue is *bidirectional*.
Data Fetching	Data can be fetched in FIFO, First In First Out manner.	Data can be fetched in *any order*.
Priorities	Priorities are *not* present in pipes.	A message can have a *priority* by attaching a priority number to type(s) of the message(s).
Receiver	For a pipe to function, sender and receiver processes should be *present* to wait for messages to be written and read in a pipe.	In a message queue, a writer process can write a message and exit. A reader process can read a message *later*.
Persistence	A pipe is deleted from the system if there is no linked receiver/sender process is present.	A message queue *remains* active in the system until explicitly deleted by some process.
Message Size	A pipe message size can be up to *4096* bytes. *(Currently it has been expanded).*	A message queue message size can be up to *8192* bytes. *(Currently it has been expanded).*

Writing Message Queue

- https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_message_queues.htm

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg)
```

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

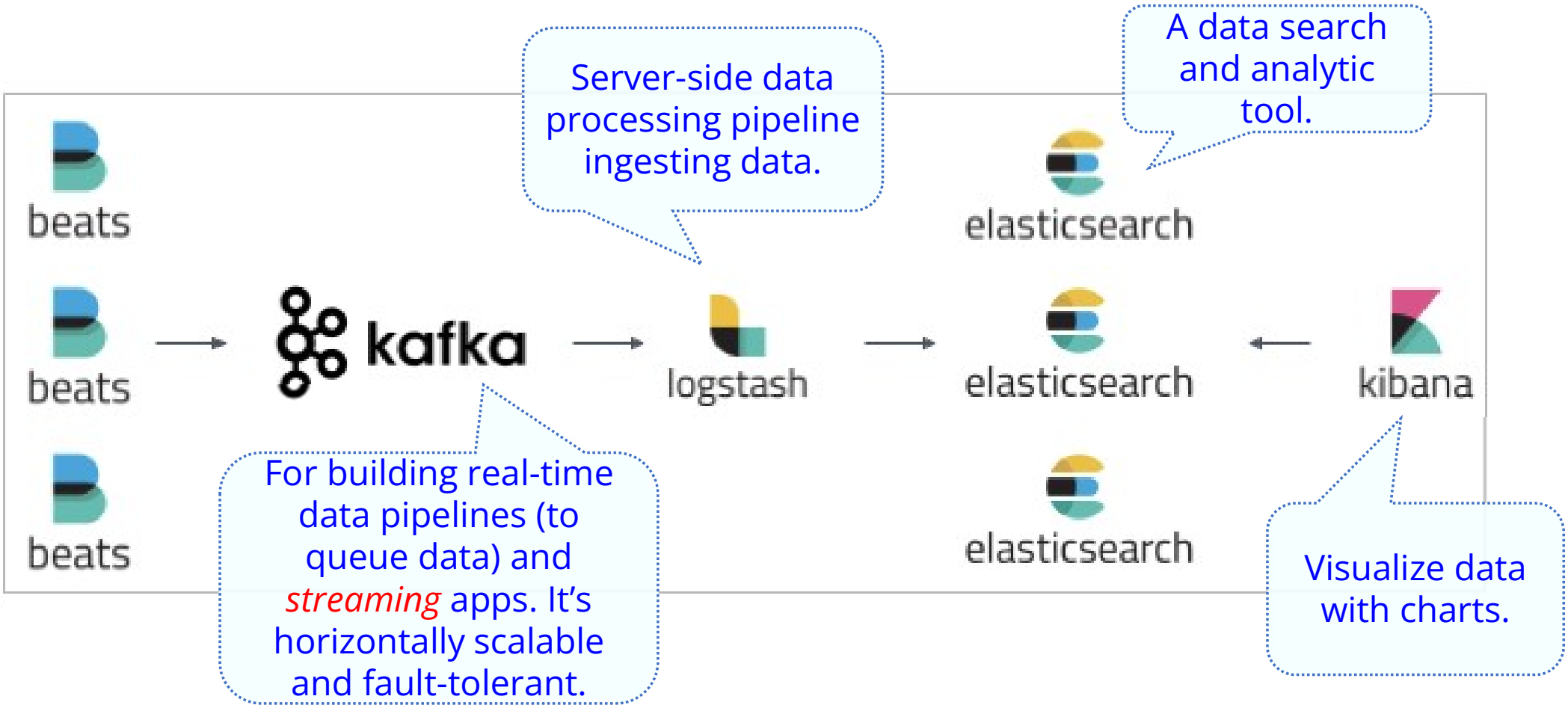
```
int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg)
```

```
int msgrcv(int msgid, const void *msgp, size_t msgsz, long msgtype, int msgflg)
```

```
int msgctl(int msgid, int cmd, struct msqid_ds *buf)
```

Get Q Info, Get Msg Info, Set ID and permissions, Remove MsgQ

Current distributed *message passing architecture – ELK stack*



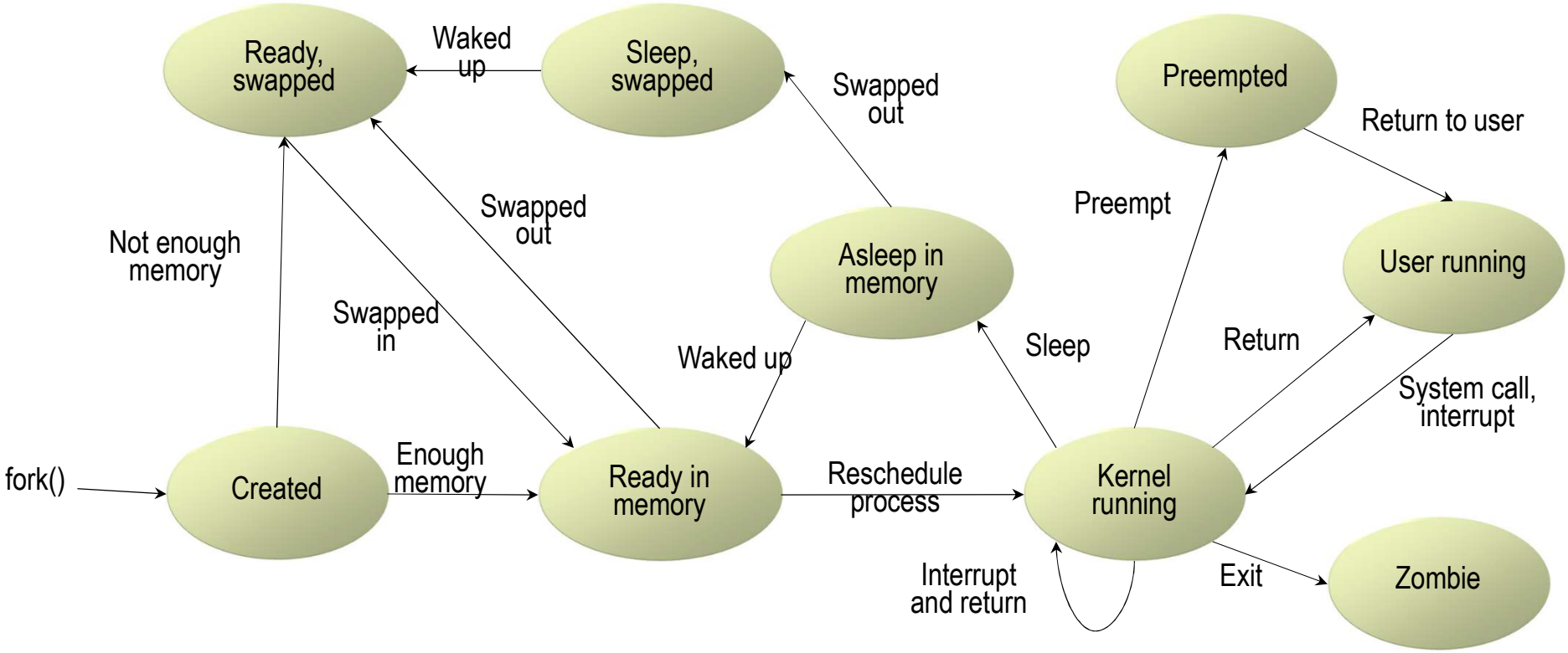
More at: https://www.elastic.co/what-is/elk-stack

Contents

- What is a process?
- Inside a process
- Management of processes
- Inter process communications (IPC)
- A brief view of processes in Unix and Android

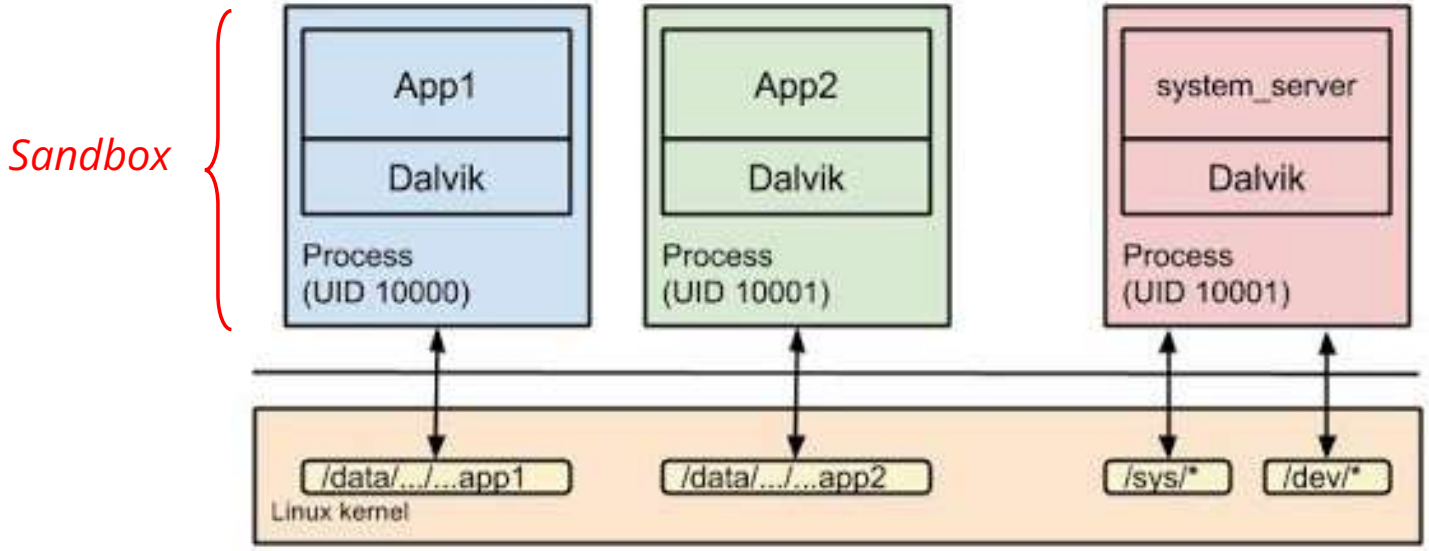
- What is a thread and its benefits?
- Management of threads
- Thread Libraries
- Programming of server-client communication

Process Life Cycle in SVR4 Unix



Applications in Android

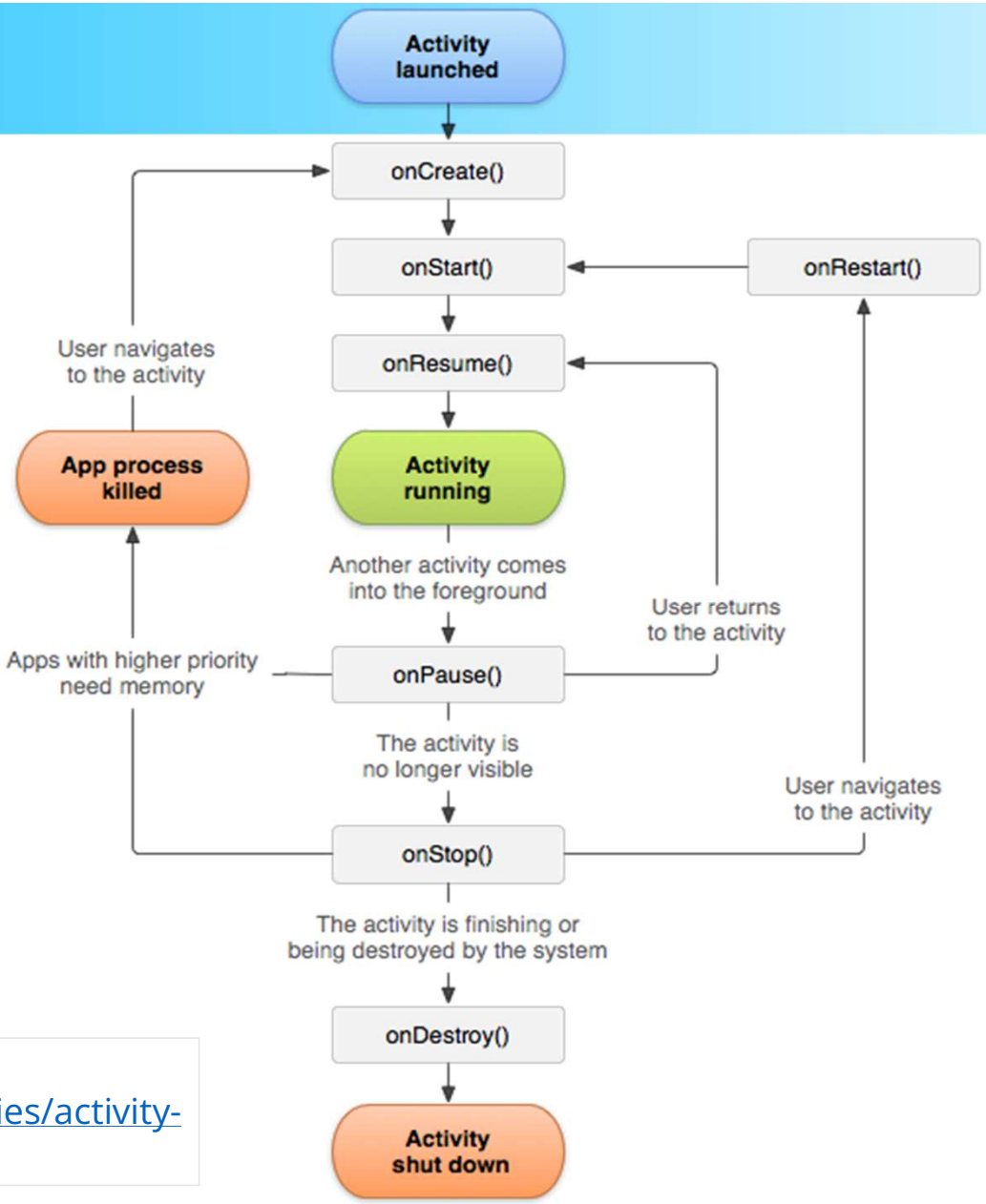
- The Android SDK tools compile your code along with any data and resource files into an APK, an Android *package*, an archive file with an .apk suffix.
- Each Android app runs in a *security sandbox*.
  - Its own process, unique Linux user ID, to control access permissions.
  - Its own VM in isolation from other apps.



Source: http://hiqes.com/android-security-part-1/

Android Activity Lifecycle

- When an application component starts, the Android system starts a new Linux process for the application with a single thread of execution, called '*main*' thread.

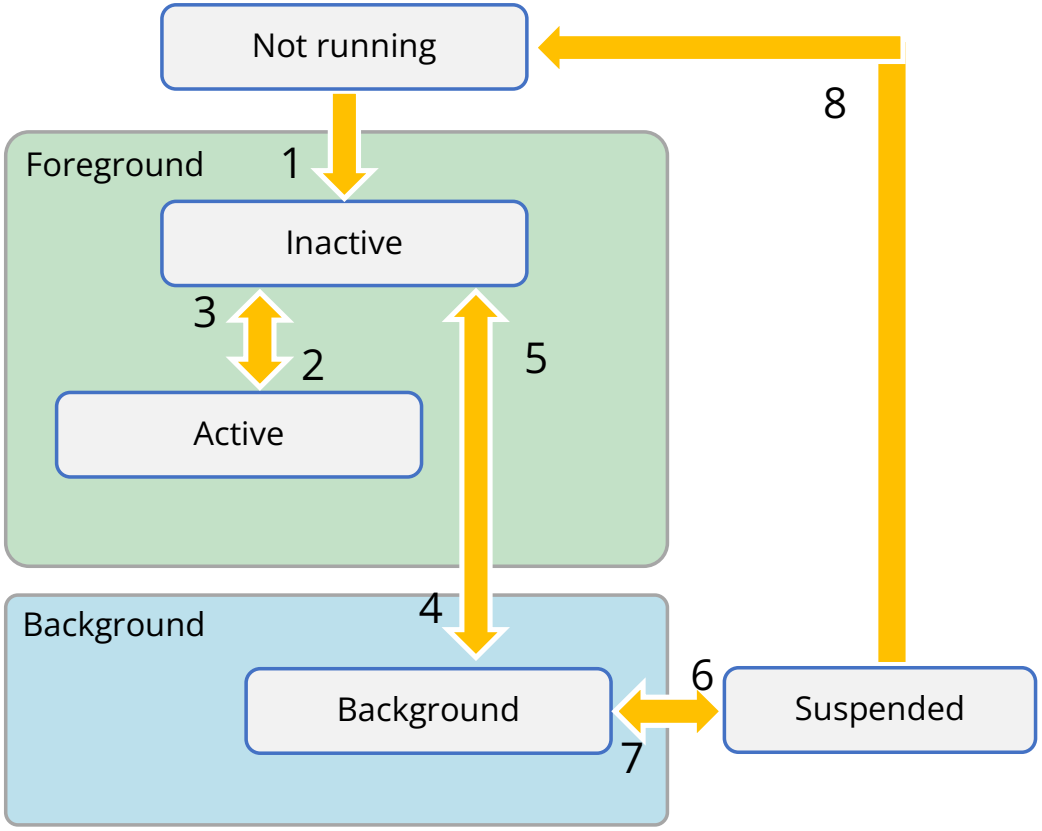


- **onPause()**- Screen is partially covered by other new activity. The Activity is not moved to Back Stack.
- **onPause() + onStop()**- Screen is fully covered by other new activity. The Activity is moved to Back Stack.

Source:
https://developer.android.com/guide/components/activities/activity-lifecycle

iOS Application life cycle

1. Not running — app wasn't launched yet or system shut it down.
2. Inactive — Application is running but it's performing another actions, it is not ready to work with user interactions.
3. Active — normal state for app that is *running* in foreground and cooperates with user. Application receives events, updates UI.
4. Background — state when application is still *running* but its interface is *not visible* for user.
5. Suspended — application is still in device memory, but the code isn't running. If a low memory condition occurs system can kill application without notice.



Contents

- What is a process?
- Inside a process
- Management of processes
- Inter process communications (IPC)
- A brief view of processes in Unix and Android

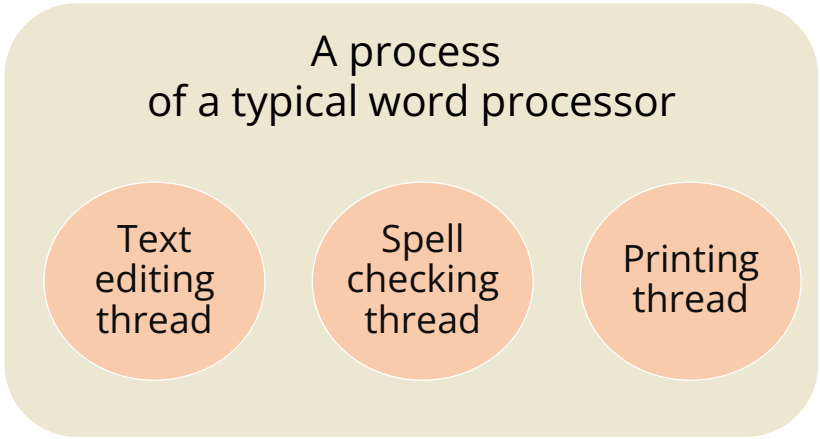
- What is a thread and its benefits?
- Management of threads
- Thread Libraries
- Programming of server-client communication

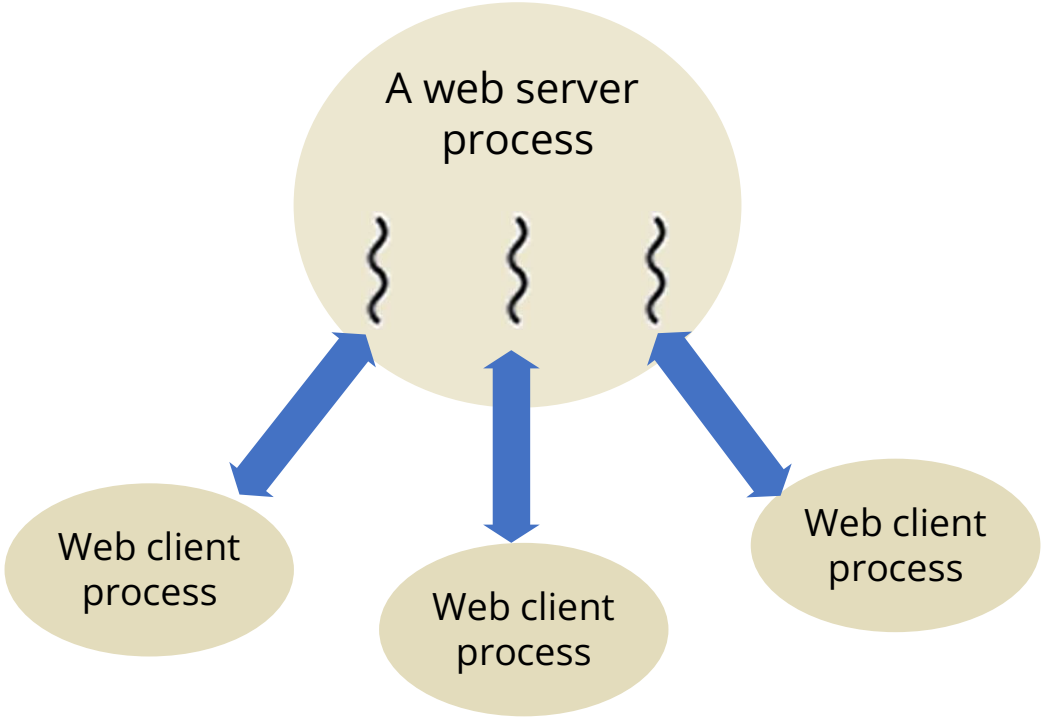
Let’s thinking

- How can we edit text in a word processor while it is checking spelling and grammar as well as printing?
- How can a web server handle multiple clients at the same time?

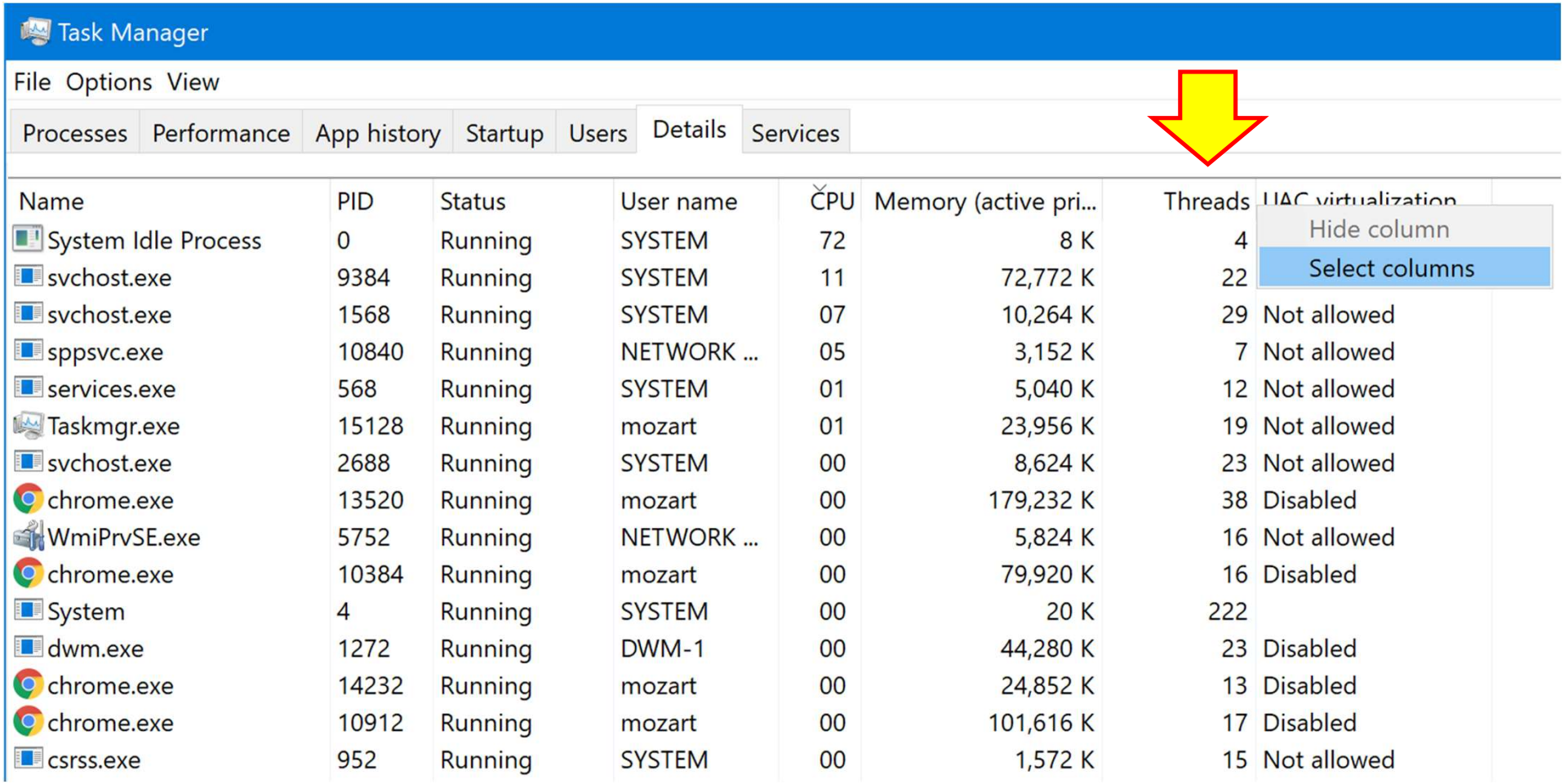
Multithreading

- A process can have more than 1 thread, being called a *multithreading* process.
- Applications of a multithreading system:
  - Word processor
  - Web server
  - Web browser, etc.



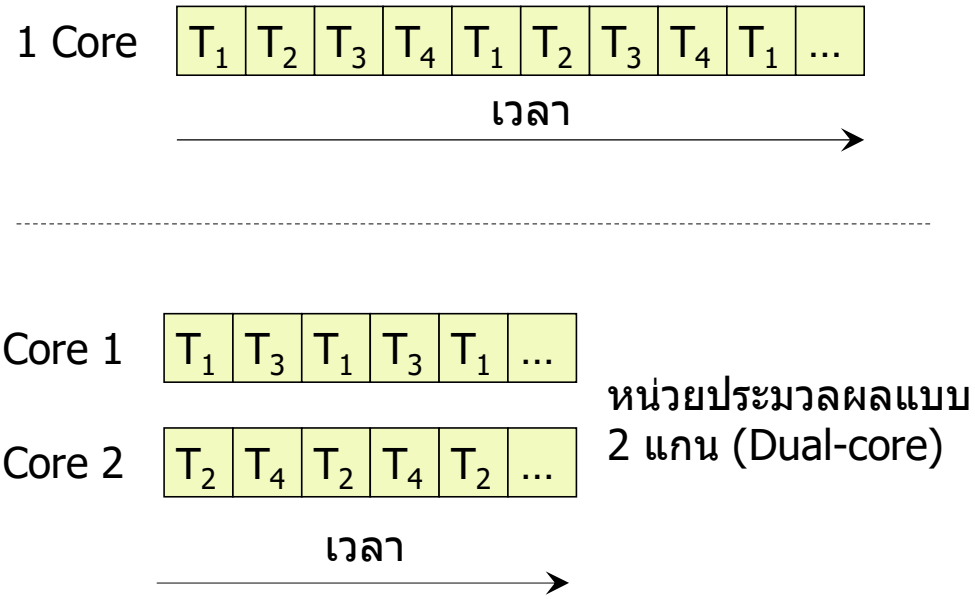


Windows's Task Manager: No. of Threads



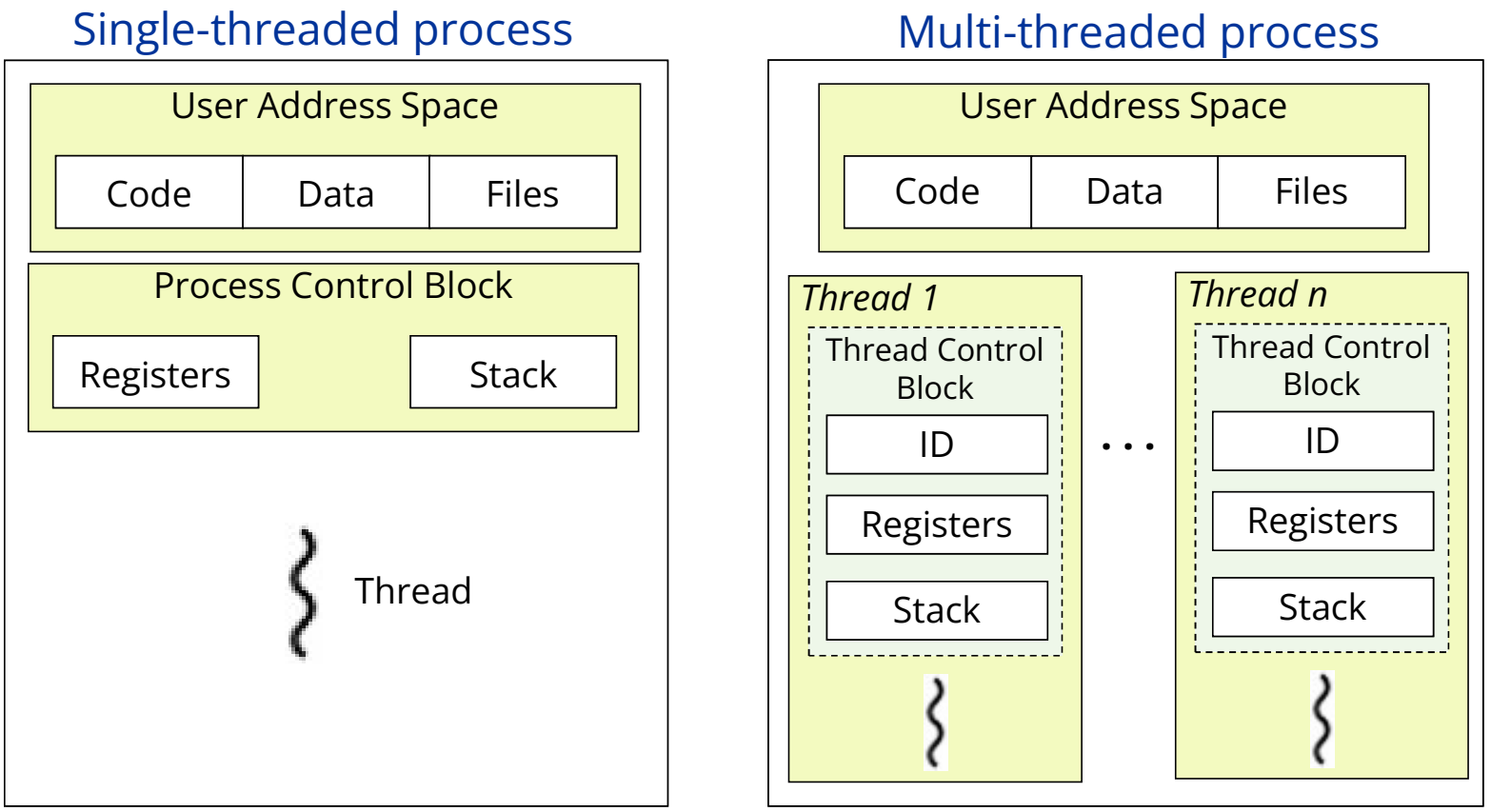
Benefits of Multithreading

- Responsiveness
- Resource sharing
- Economical
- More efficiency on multicore processor



Characteristics of a thread

- Basic components of a thread: Thread ID, Program Counter, Registers, and Stack

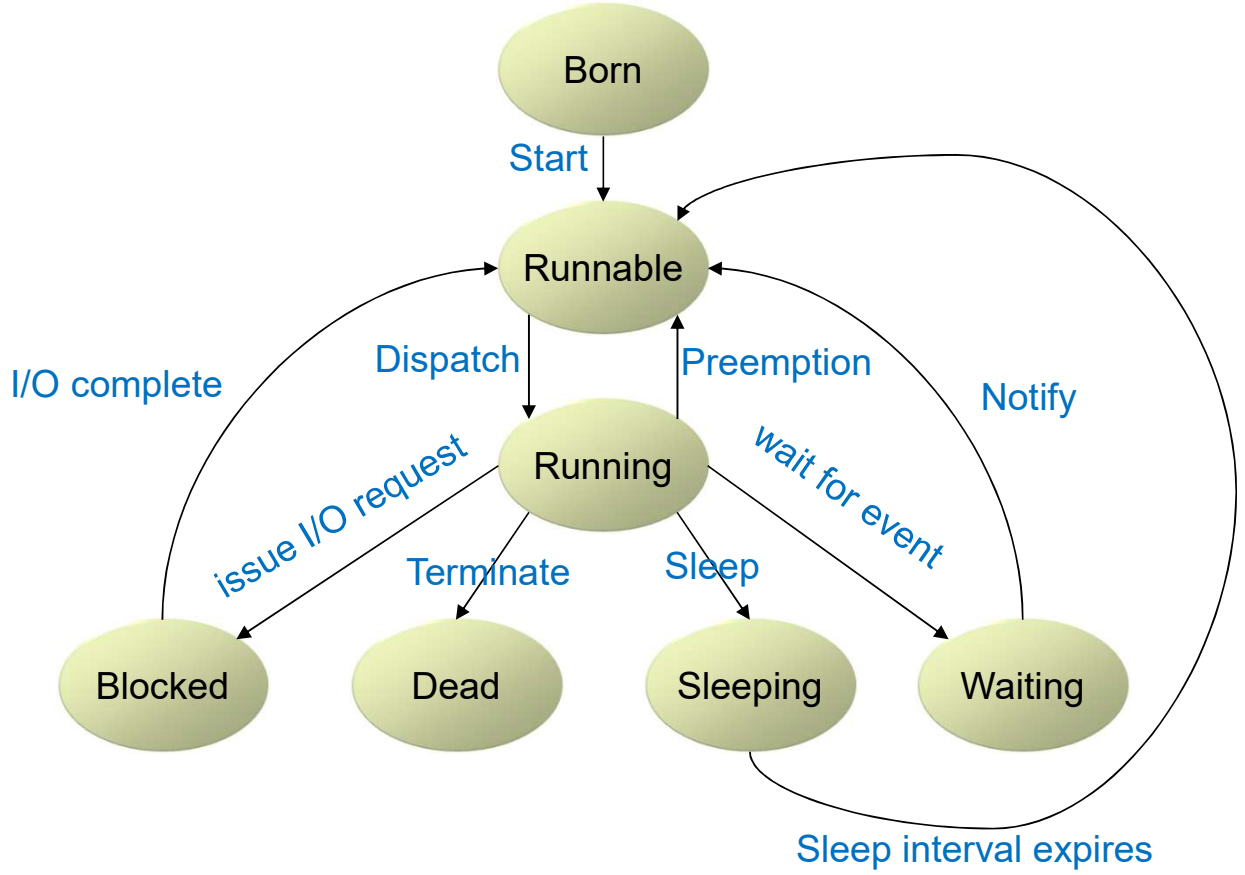


Contents

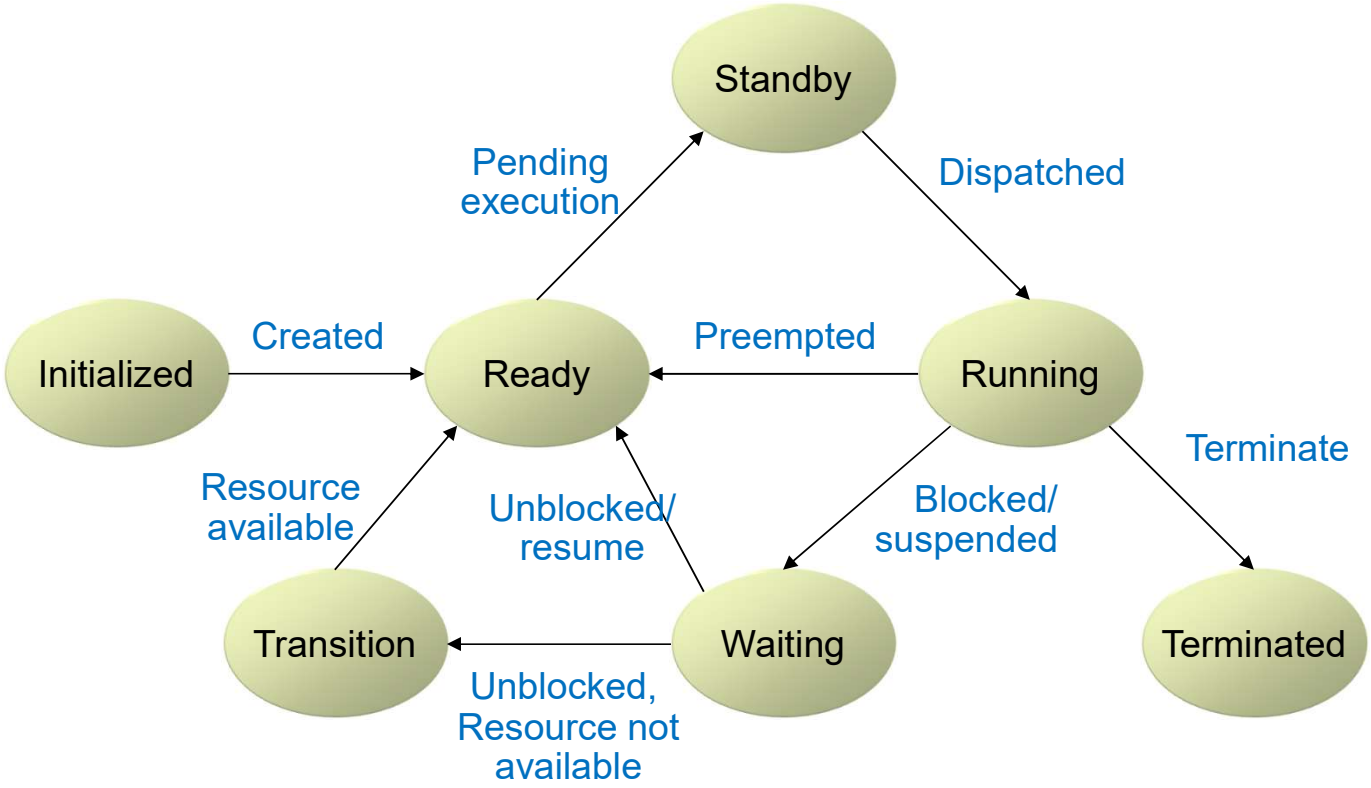
- What is a process?
- Inside a process
- Management of processes
- Inter process communications (IPC)
- A brief view of processes in Unix and Android

- What is a thread and its benefits?
- Management of threads
- Thread Libraries
- Programming of server-client communication

States of a Thread: Java Thread



Windows Threads Status



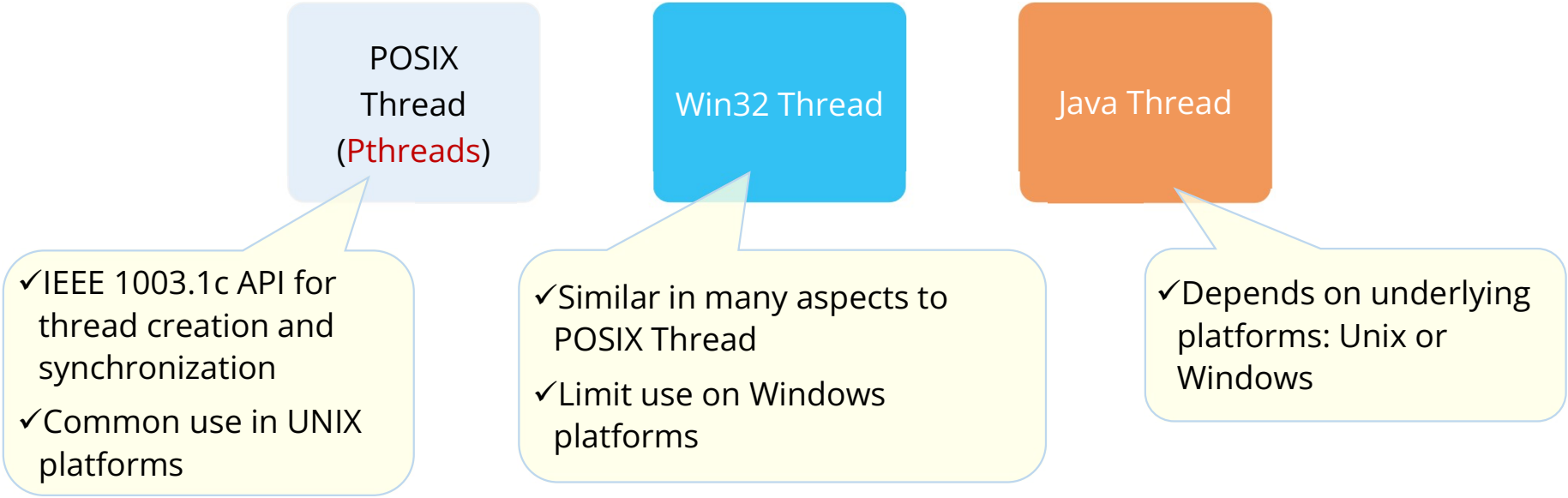
Contents

- What is a process?
- Inside a process
- Management of processes
- Inter process communications (IPC)
- A brief view of processes in Unix and Android

- What is a thread and its benefits?
- Management of threads
- Thread Libraries
- Programming of server-client communication

Thread Libraries

- Thread library is a set of API for creating and managing threads.
- Three notable thread libraries today are:

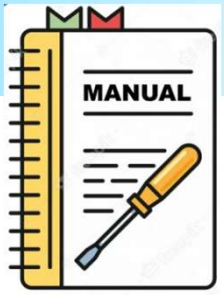


POSIX (Portable Operating Systems Interface for Computing Environments)
by IEEE's Portable Application Standards Committee

Pthreads (1/2)

```
1.#include <unistd.h>
2.#include <stdio.h>
3.#include <stdlib.h>
4.#include <pthread.h>
5.
6.char message[] = "Hello World";
7.void *ThreadFunction(void *arg) {
8.    printf("ThreadFunction started. Arg is %s\n", (char *)arg);
9.    sleep(3);
10.    strcpy(message, "Bye!");
11.    pthread_exit("ThreadFunction Exited.");
12.}
```

Pthreads (2/2)

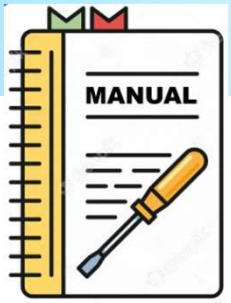



```
13.int main() {
14.     int       res;
15.     pthread_t myThread;
16.     void      *threadRes;
17.     res = pthread_create( &myThread, /* returned ID */
18.                NULL,                 /* default attribute */
19.                ThreadFunction,       /* prt to thread function */
20.                (void *)message );    /* ptr to argument */
21.     if (res != 0) {
22.         perror("Thread creation failed");
23.         exit(EXIT_FAILURE);
24.     }
25.     printf("Waiting for thread to finish...\n");
26.     res = pthread_join(myThread, &threadRes);
27.     if (res != 0) {
28.         perror("Thread join failed");
29.         exit(EXIT_FAILURE);
30.     }
31.     printf("Thread joined, it returned %s\n", (char *)threadRes);
32.     printf("Message is now %s\n", message);
33.     exit(EXIT_SUCCESS);
34.}
```

Win32 Thread (1/2)

```
1.#include <windows.h>
2.#include <stdio.h>
3.#include <stdlib.h>

4.char message[] = "Hello World";

5.DWORD WINAPI ThreadFunction(PVOID arg) {
6.    printf("ThreadFunction started. Arg was %s\n", (char *)arg);
7.    Sleep(3000);
8.    strcpy(message, "Bye!");
9.    return 100;
10.}

11.void main() {
12.    HANDLE a_thread;
13.    DWORD myThreadID;
14.    DWORD threadRes;
```

Win32 Thread (2/2)




```
15.     /* Create a new thread. */
16.     a_thread = CreateThread(NULL, /* default security attribute */
17.                 0,                /* default thread stack size */
18.                 ThreadFunction,   /* thread function to run */
19.                 (PVOID)message,   /* parameter passed to thread */
20.                 0,                /* default creation flag */
21.                 &myThreadID);     /* returned thread ID */
22.     if (a_thread == NULL) {
23.         perror("Thread creation failed");
24.         exit(EXIT_FAILURE);
25.     }
26.     printf("Waiting for thread to finish...\n");
27.     if (WaitForSingleObject(a_thread, INFINITE) != WAIT_OBJECT_0) {
28.         perror("Thread join failed");
29.         exit(EXIT_FAILURE);
30.     }

31.     /* Retrieve the code returned by the thread. */
32.     GetExitCodeThread(a_thread, &threadRes);
33.     printf("Thread joined, it returned %d\n", threadRes);
34.     printf("Message is now %s\n", message);
35.     exit(EXIT_SUCCESS);
36. }
```

Java Thread (1/2)

```
1./* To compile, enter javac MainThread.java   */
2.class Parameter
3.{
4.  private String val;
5.  public String get() {
6.    return val;
7.  }
8.  public void set(String val) {
9.    this.val = val;
10.  }
11.}
12.class MyThread implements Runnable
13.{
14.  private Parameter msg;
15.    public MyThread(Parameter m) {
16.                  msg = m;
17.    }
```

Java Thread (2/2)

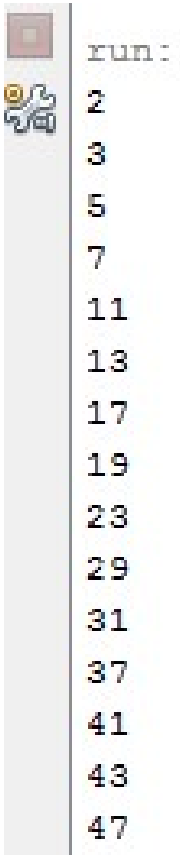
```
18.     public void run() {
19.         System.out.println("Thread starts. Message is " + msg.get());
20.         msg.set("Bye!");
21.     }
22.}

23.public class MainThread
24.{
25.    public static void main(String[] args) {
26.       Parameter message = new Parameter("Hello World");
27.       Thread worker = new Thread(new MyThread(message));
28.       worker.start();
29.       try {
30.          worker.join();
31.       } catch (InterruptedException ie) { }
32.       System.out.println("Message is now " + message);
33.    }
34.}
```

Multithreading Prime Number in Java

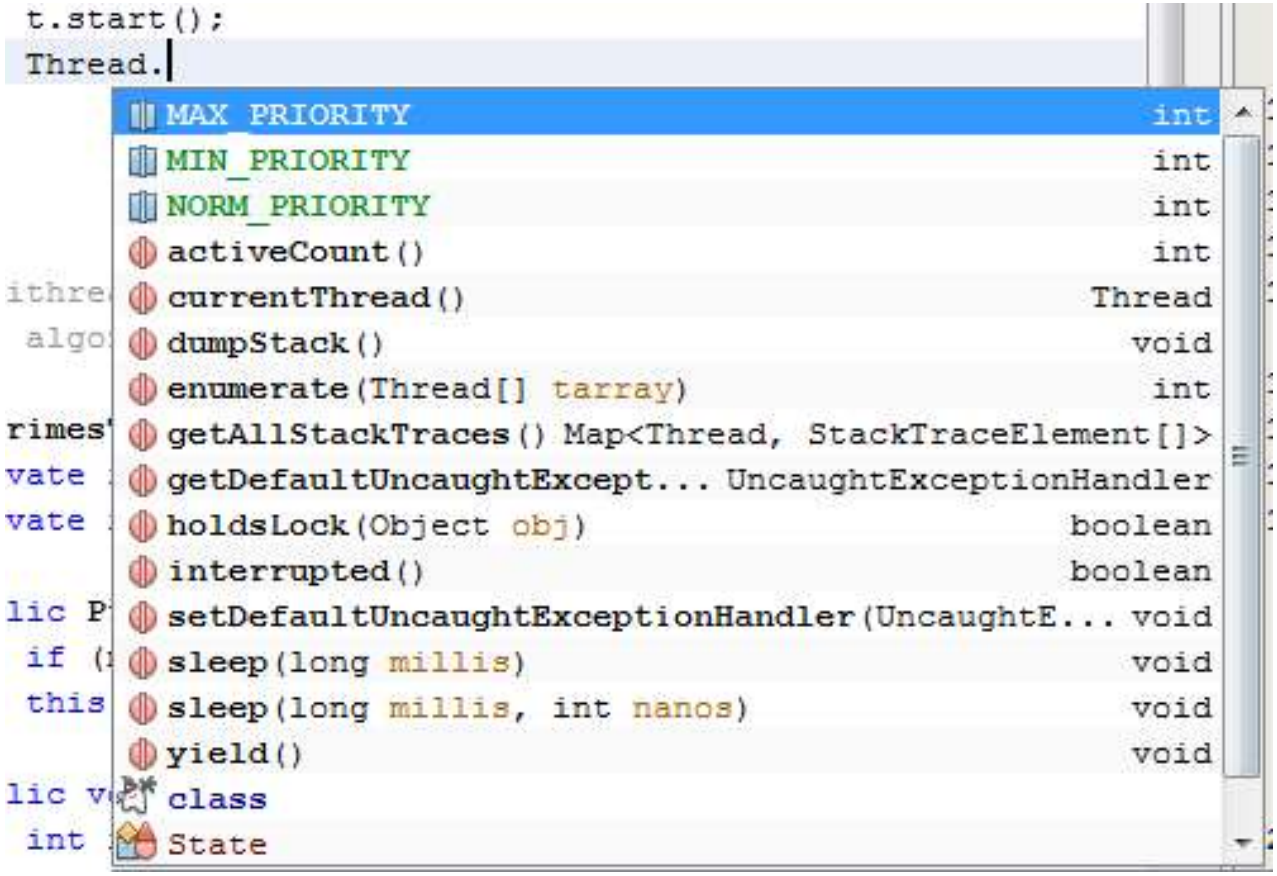
```
public class Primes {
	public static void main(String args[]) {
		int limit;
		if (args.length == 0) limit = 50;
		else limit = Integer.parseInt(args[0]);
		Thread t = new Thread(new PrimesThread(limit));
		t.start();
	}
}
/** Multithreaded prime number generator in Java. */
class PrimesThread implements Runnable {
	private int num;
	private int[] primeNums;

	public PrimesThread(int num) {
		if (num < 2) throw new IllegalArgumentException();
		this.num = num;
	}
```

Threaded Prime in Java

```
18.     public void run() {
19.             int i, j;
20.             primeNums = new int[num + 1];
21.             primeNums[1] = 0;
22.             for (i = 2; i <= num; i++)
23.                     primeNums[i] = 1;
24.             for (i = 2; i <= num/2; i++)
25.                     for (j = 2; j <= num/i; j++)
26.                             primeNums[i*j] = 0;
27.             for (i = 1; i <= num; i++)
28.                     if (primeNums[i] > 0)
29.                             System.out.println(i);
30.     }
31.}
```



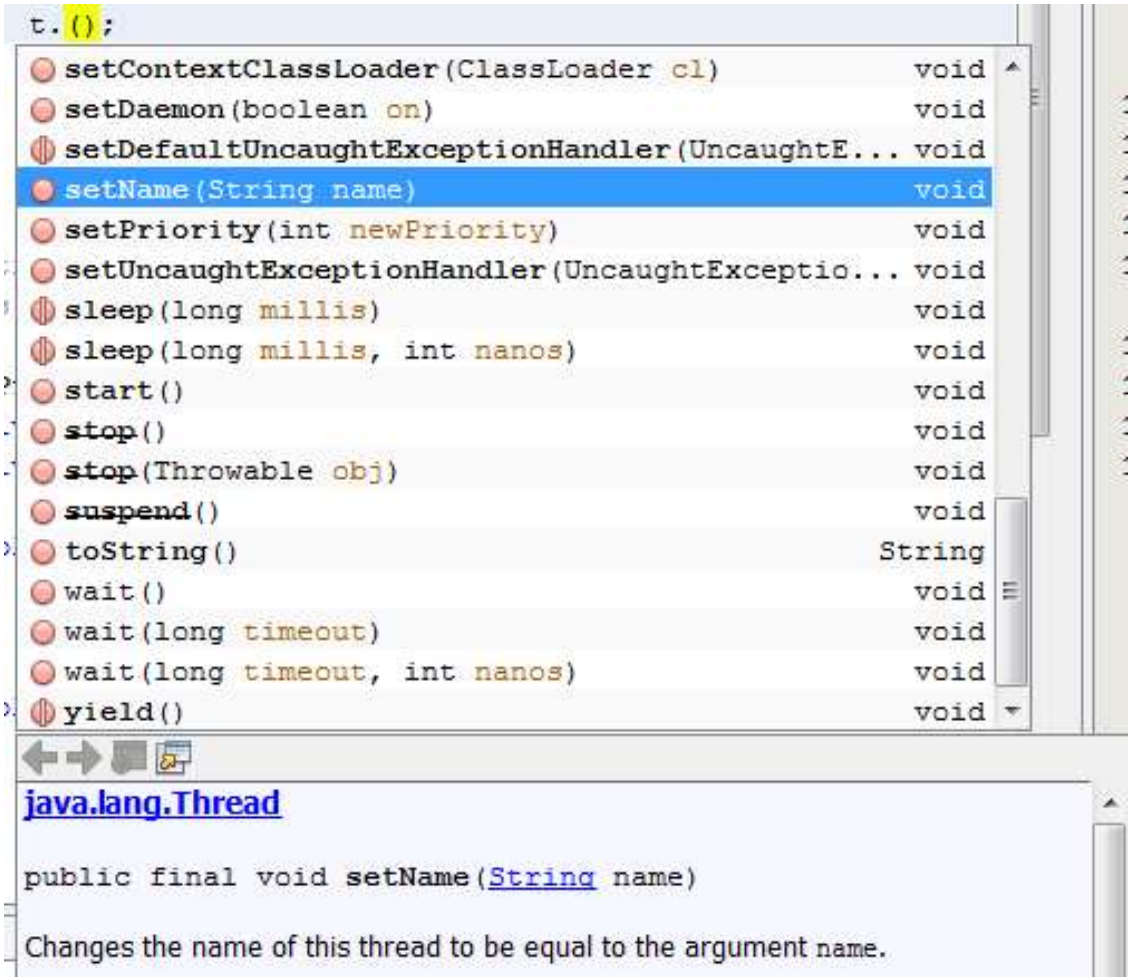
Members of Java's class Thread



More: https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

Members of Java's class Thread

```
.getID()
.getName()

.setName()
.setPriority()

.toString()

.start()
.sleep()
.yield()

etc.
```



Demo codes

- D:\courses\CSS226-OS\codes\osLabs\src\BasicThreads
  - ✓Primes.java
  - ✓SimpleThreadSolution.java

Assignments

2. Modify previous program of Java Prime to have it create and simultaneously run 4 (or other number of) threads, given by user. Each run will sleep a random number of seconds after printing each line. Note to show thread ID/Name on every output line.

- ***Hints:***
- `import.util.Random;`

- `Random r = new Random();`
- `Thread myT = ...;`

- `myT.sleep(r.nextInt() % 4000);`

- Use methods `setName()`, `getName()`, `getId()`, etc.

Contents

- What is a process?
- Inside a process
- Management of processes
- Inter process communications (IPC)
- A brief view of processes in Unix and Android

- What is a thread and its benefits?
- Management of threads
- Thread Libraries
- Programming of server-client communication

Socket programming

- Sockets are communication *endpoints* on the *same or different* computers to exchange data.
- The concept and API of *Socket programming* were developed for C on Unix since 1970s.
- Currently supported on Windows, Mac, and many other OS's with C, Java, Python and many other languages.

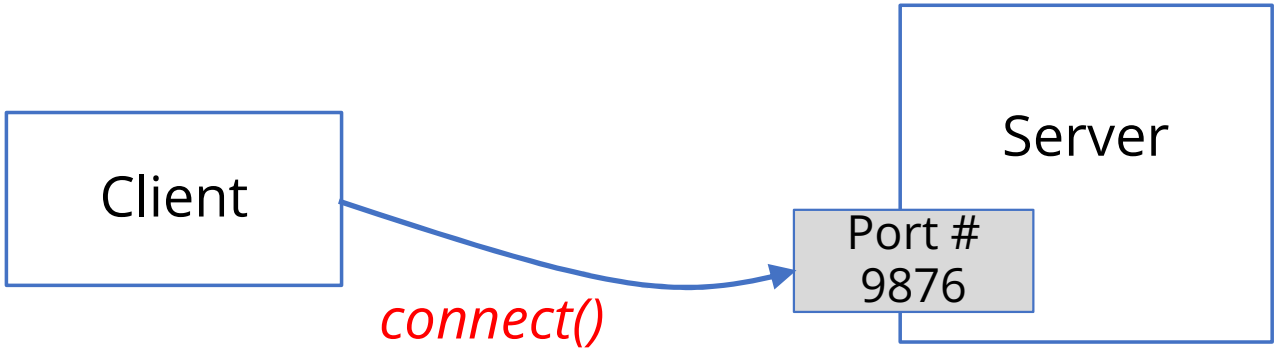
Socket programming for client-server architecture

- *Client* process typically makes a request for information. After getting the response, this process may do some other processing.
  - Example: web browser
- *Server* process takes a request from the clients. After getting a request from the client, this process will perform the requested processing, gather the result information, and send it to the requestor client. Once done, it becomes ready to serve another client.
  - Example: web server
- The client needs to know the address of the server, but the server does not need to know the address or even the existence of the client prior to the connection being established. Once a connection is established, both sides can send and receive information.

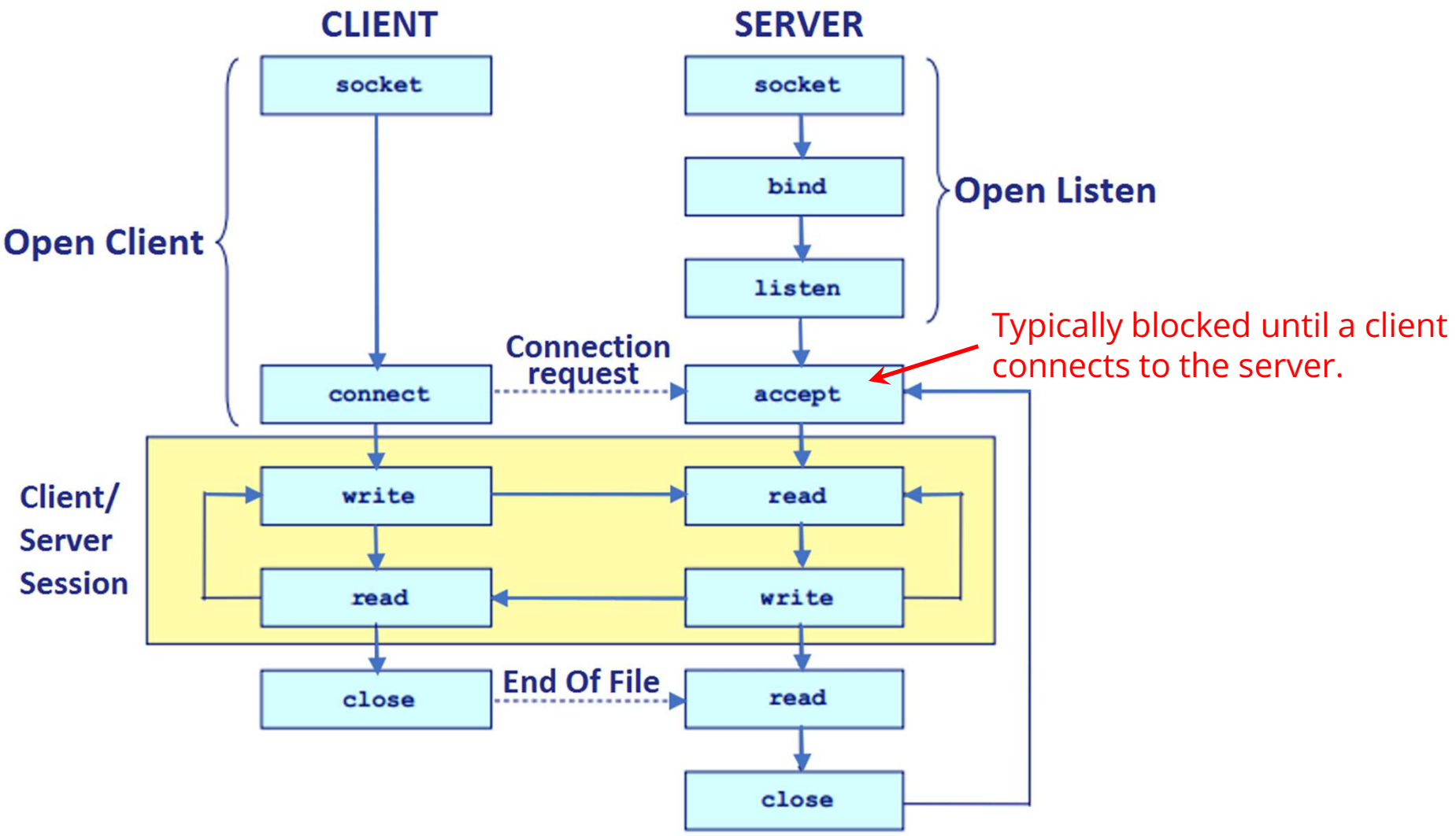
Main types of server

- Iterative Server
  - Simple
  - a server process serves one client and after completing the first request, it takes request from another client. Meanwhile, another client keeps waiting.
- Concurrent Server
  - runs multiple concurrent processes (or threads) to serve many requests at a time because one process may take longer and another client cannot wait for so long.
  - The simplest way to write a concurrent server under Unix is to fork a child process to handle each client separately, or to create a thread for each incoming client.

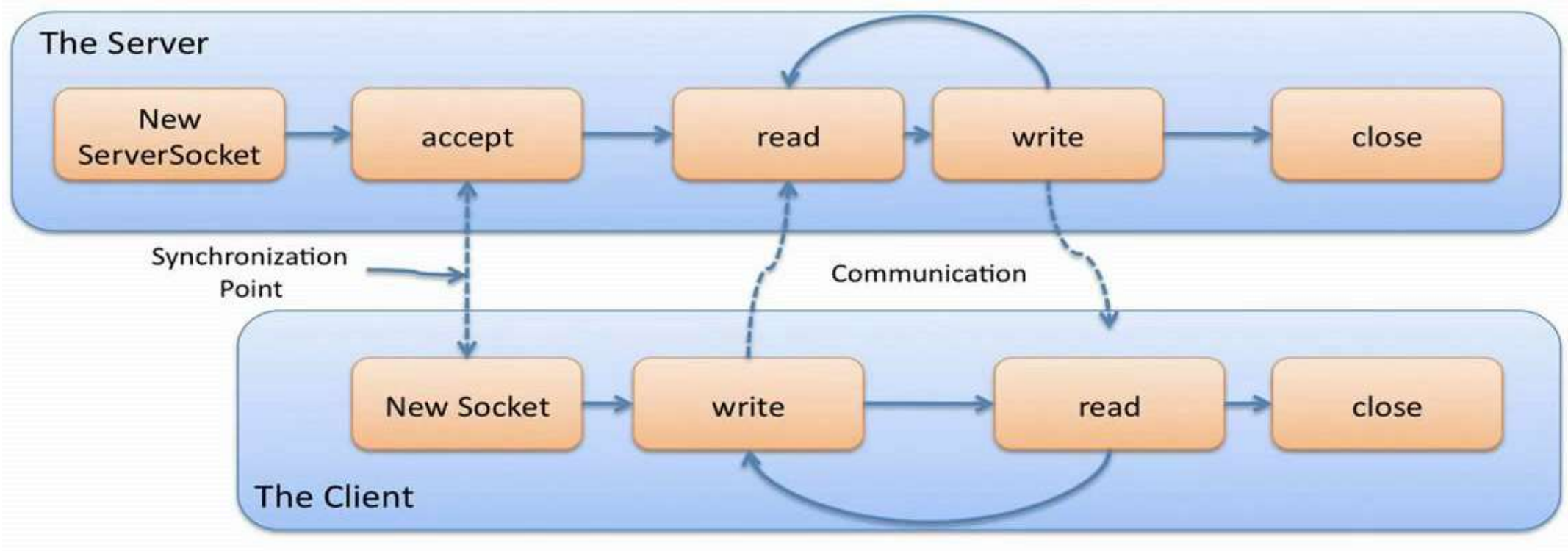
Basic architecture of socket-based client-server



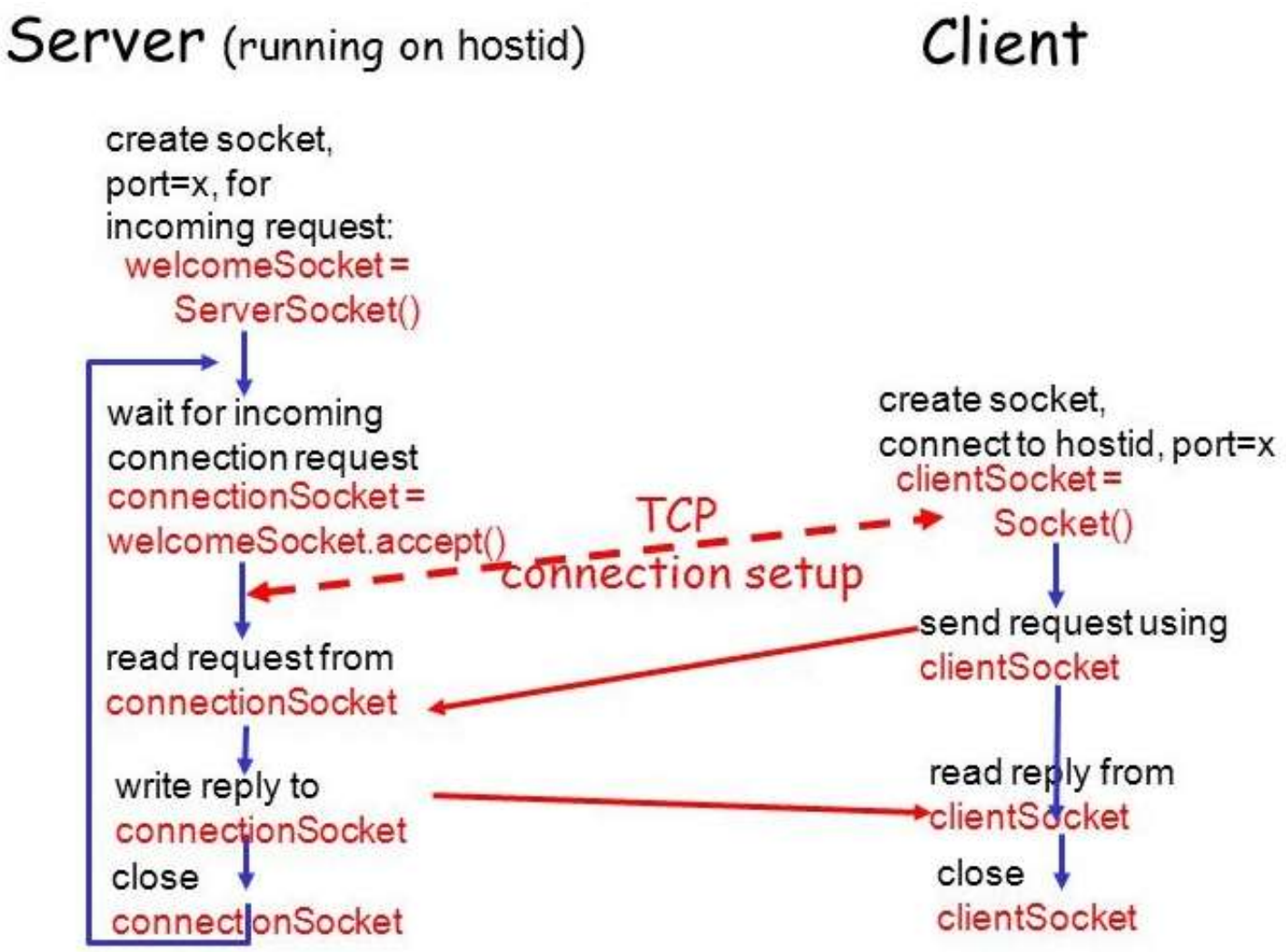
Unix/Linux C-based Socket API



Java-Based Socket Overview



Java-Based Server-Client Communication



Java-Based Server-Client Communication

Server.java

Source | History

```
1  import java.io.*;
2  import java.net.*;
3
4  public class Server {
5
6      public static void main(String[] args) {
7          try {
8
9              // create a socket at port # 6789
10             ServerSocket ss = new ServerSocket(6789);  // create a socket
11             System.out.println("A socket is created and now waiting for connection.");
12
13             // establish and wait for an incoming connection
14             Socket s = ss.accept();
15             System.out.println("A client has made a connection in.");
16
17             DataInputStream din = new DataInputStream(s.getInputStream());
18             DataOutputStream dout = new DataOutputStream(s.getOutputStream());
19
20             // wait for input message and display it
21             System.out.println("Waiting for an incoming message...");
22             String str = (String) din.readUTF();
23             System.out.println("Message received: " + str);
24
25             str = str + " " + str;
26             System.out.println("Now I'm going to write this message back:" + str);
27             dout.writeUTF(str);
28
29             ss.close();
30
31         } catch (IOException e) { System.out.println(e); }
32     }
33 }
```

Server

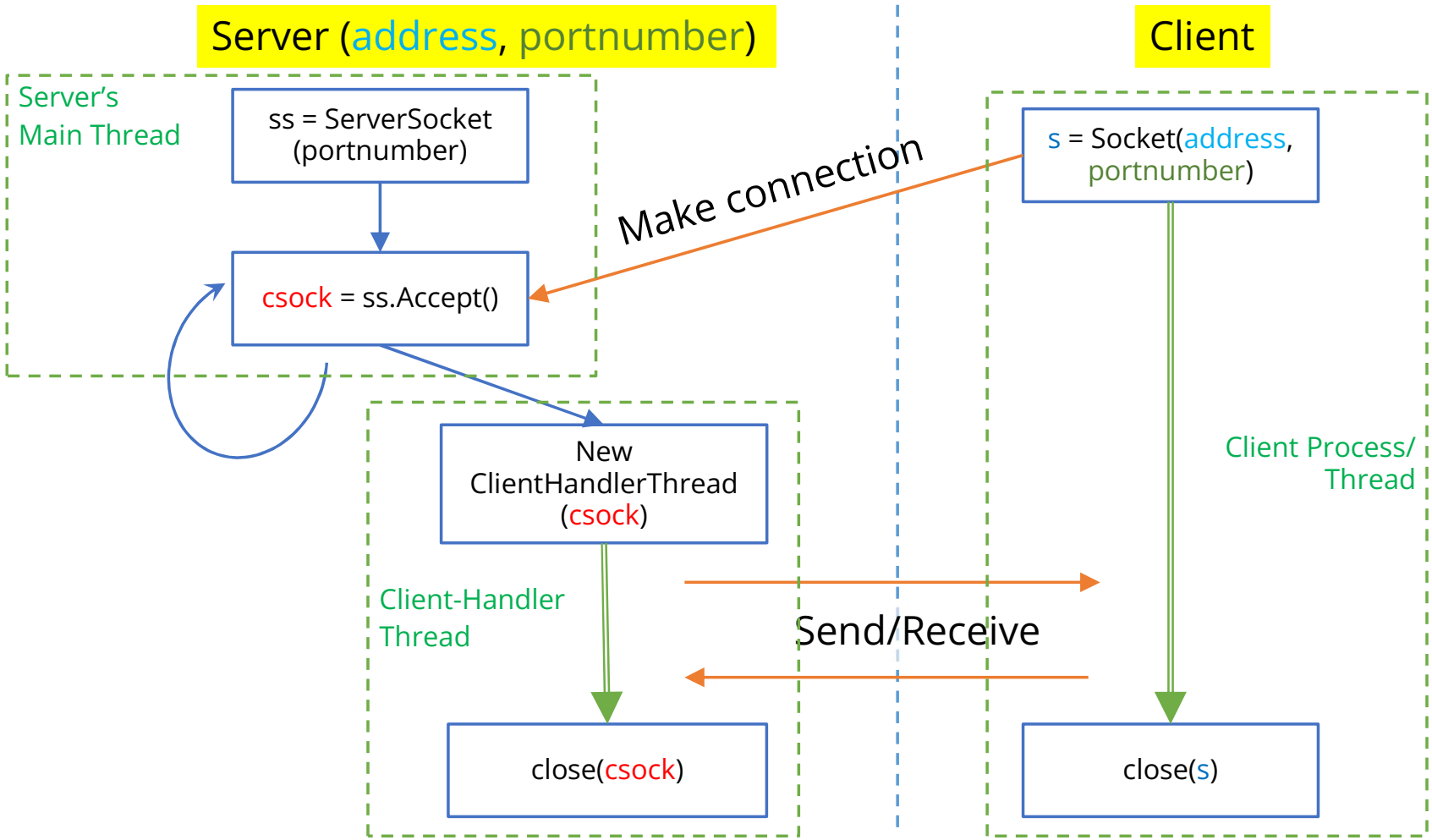
Client.java

Source | History

```
1  import java.io.*;
2  import java.net.*;
3
4  public class Client {
5
6      public static void main(String[] args) {
7          try {
8              // create a socket to a local host with port # 6789
9              Socket s = new Socket("localhost", 6789);
10
11             System.out.println("A connection is established and I'll now
12
13             DataOutputStream dout = new DataOutputStream(s.getOutputStrea
14             DataInputStream din = new DataInputStream(s.getInputStream())
15
16             //Thread.sleep(5000);
17             dout.writeUTF("Hello Server");
18             dout.flush();
19
20             String str = (String) din.readUTF();
21             System.out.println("Message received from server: " + str);
22
23             dout.close();
24             s.close();
25
26         } catch (IOException e) { System.out.println(e); }
27     }
28 }
29
```

Client

Workflow for a multi-client server



Demo codes

- D:\courses\CSS226-OS\codes\osLabs\src\Socket
  - ✓Server.java
  - ✓Client.java
  - ✓MultiClientServer folder
    - ✓ Server.java
    - ✓ Client.java

Alternatives to Socket

- Socket programming is a low-level API for network communication and can be used with any network protocol.
- *WebSockets*: bi-directional communication between a client and server over a single, long-lived connection.
  - Real-time applications such as chat applications, online games, and stock tickers.
  - Designed to work well with web browsers.
  - Provide a secure communication channel using encryption.
- *REST APIs*: Representational State Transfer (REST) for building web services.
  - Typically use HTTP and return data in a structured format, such as JSON or XML.
  - Typically used for CRUD (Create, Read, Update, Delete) operations.
  - Higher latency due to the overhead of establishing and tearing down a connection for each request.
  - Easier to scale and be load balanced than WebSockets.

Assignment 2.1 – Process and Communication

- เขียนโปรแกรมภาษา C บน Linux เพื่อให้สร้าง (fork()) โพรเซสลูกออกมา จากนั้น ให้โพรเซสผู้สร้าง (เรียกว่าโพรเซสแม่) อ่านไฟล์ข้อความ (text file) หนึ่งที่ยาวพอสมควร เรียกฟังก์ชัน word_count() เพื่อนับคำครั้งละ 1 ย่อหน้า แสดงจำนวนคำออกมาบนหน้าจอ แล้วส่งข้อความย่อหน้านั้นให้โพรเซสลูกอย่างต่อเนื่องครั้งละ 1 ย่อหน้า จนกว่าจะสิ้นสุดไฟล์ข้อความนั้น หากส่งข้อความไปโพรเซสลูกไม่ได้ให้แจ้งข้อผิดพลาดด้วย
- ส่วนโพรเซสลูกจะวนรับข้อความย่อหน้านั้นจากโพรเซสแม่เข้ามาแล้วเรียกฟังก์ชัน word_count() เพื่อนับคำเช่นกัน และส่งผลการนับออกบนหน้าจอตัวเองเพื่อให้ตรวจสอบว่าตรงกับที่โพรเซสแม่รายงานมา ทั้งนี้สมมติว่าโพรเซสลูกทำงานช้าพอสมควรด้วยติดภาระกิจอื่นๆ มากมาย (โดยการใส่คำสั่ง sleep() ช่วงสั้น ๆ แต่บ่อย ๆ เช่น ก่อนการ read ข้อความจากโพรเซสแม่)

Summary

- What is a process?
- Inside a process
- Management of processes
- Inter process communications (IPC)
- A brief view of processes in Unix and Android

- What is a thread and its benefits?
- Management of threads
- Thread Libraries
- Programming of server-client communication