CSI3131 - Module 2: Processes

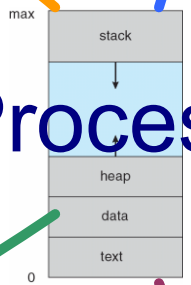
Reading: Chapter 3 (Silberchatz)

Objective: To understand the nature of processes including the following concepts:

- **Process states, the PCB (process control block), and process switching.**
- **Process scheduling**
- **Operations on processes**
- **Mechanisms for supporting cooperation between processes**

Concept

Scheduling



Process

Operations

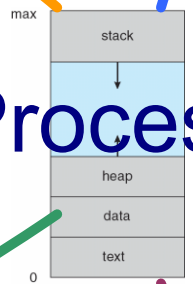
Cooperation:
IPC

Program Execution

Process State

Concept

Scheduling



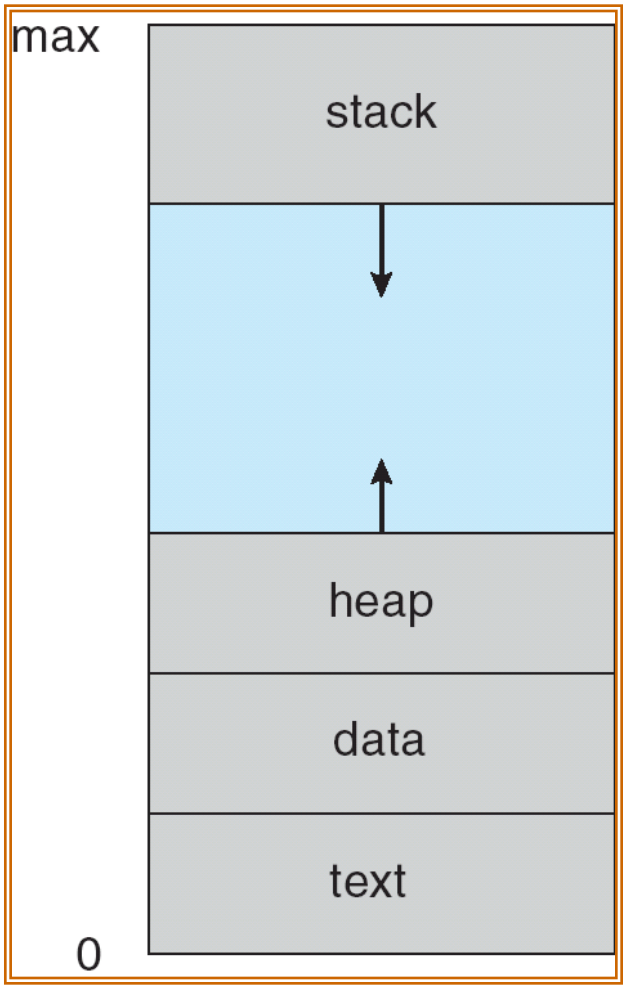
Process

Operations

Cooperation: IPC

Process Concept

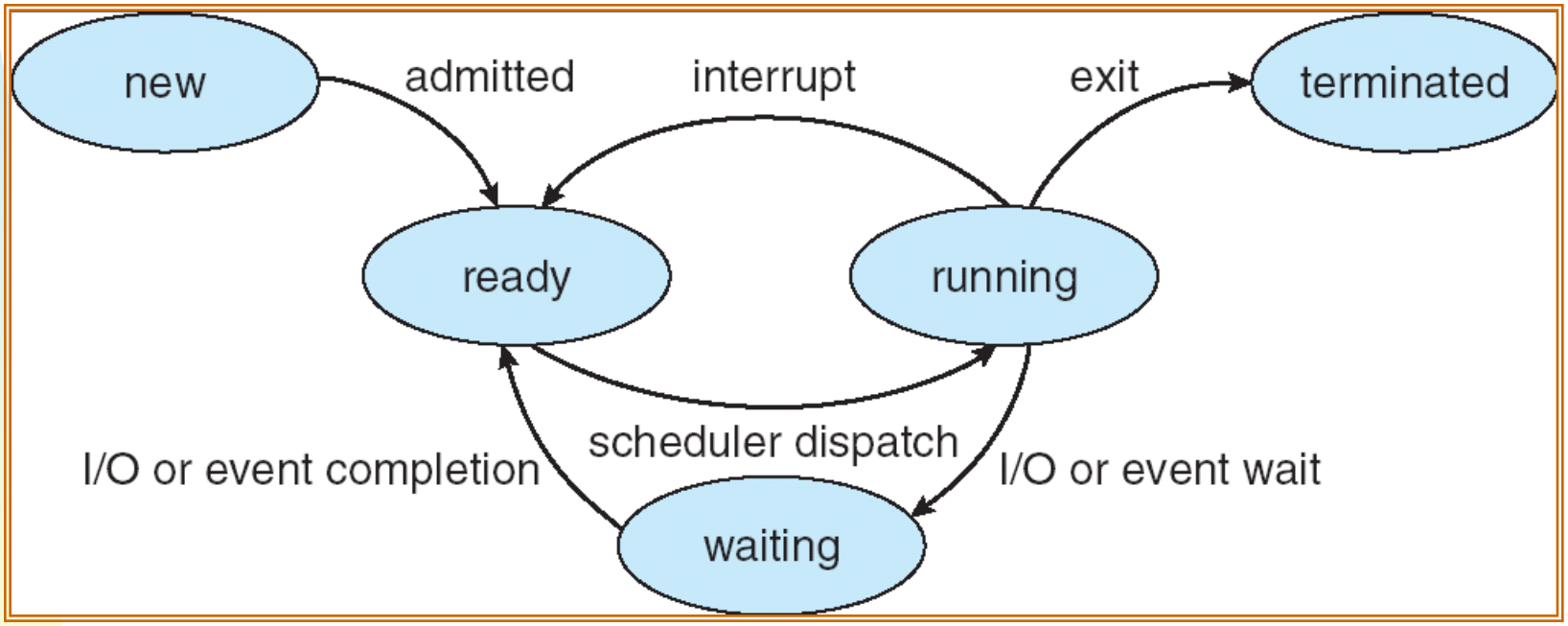
- Process executes a program
  - Text – code to execute
  - Data
  - Heap (dynamic memory)
  - Stack (local variables)
- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- We will use the terms *job*, *task*, and *process* almost interchangeably



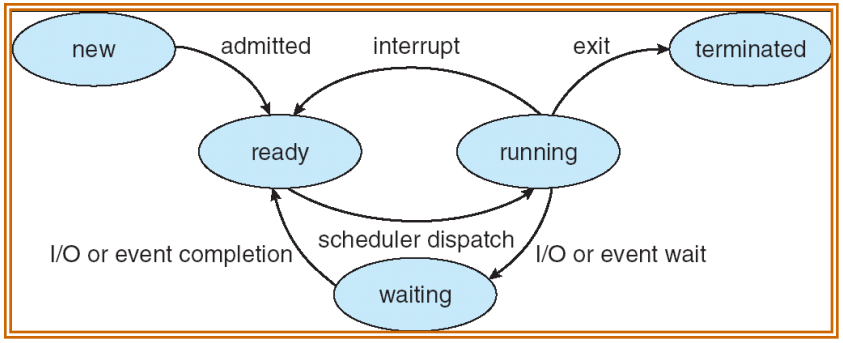
Process States

- **Which properties of process states should OS distinguish?**
  - **running vs not running**
  - **blocked/waiting vs ready**
- **Main process states:**
  - new: **The process is being created**
  - running: **Instructions are being executed**
  - waiting: **The process is waiting for some event to occur**
  - ready: **The process is waiting to be assigned to a process**
  - terminated: **The process has finished execution**
- **In real OSes, there could be additional states**
  - **i.e. has the process been swapped out?**

Simple Diagram of Process States

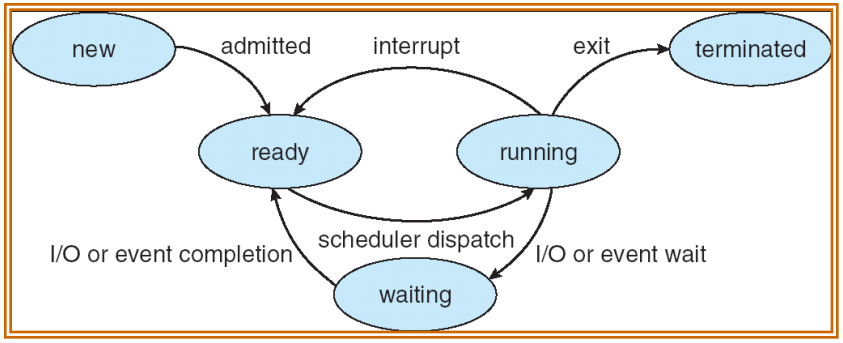


Process state transitions

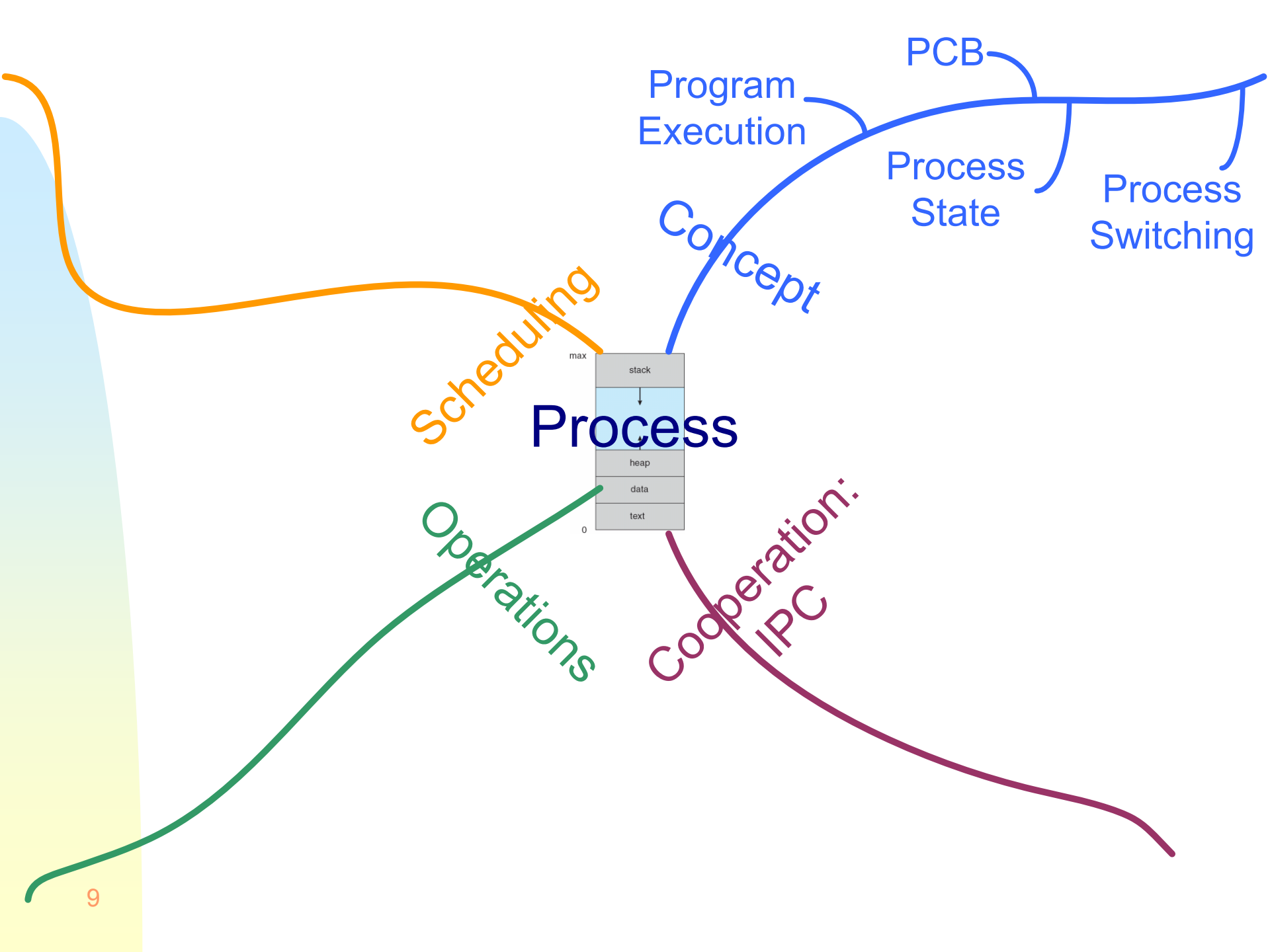


- New → Ready
  - Represents the creation of the process.
    - In the case of batch systems – a list of waiting in new state processes (i.e. jobs) is common.
- Ready → Running
  - When the CPU scheduler selects a process for execution.
- Running → Ready
  - Happens when an interruption caused by an event independent of the process
    - Must deal with the interruption – which may take away the CPU from the process
      - Important case: the process has used up its time with the CPU.

Process state transitions



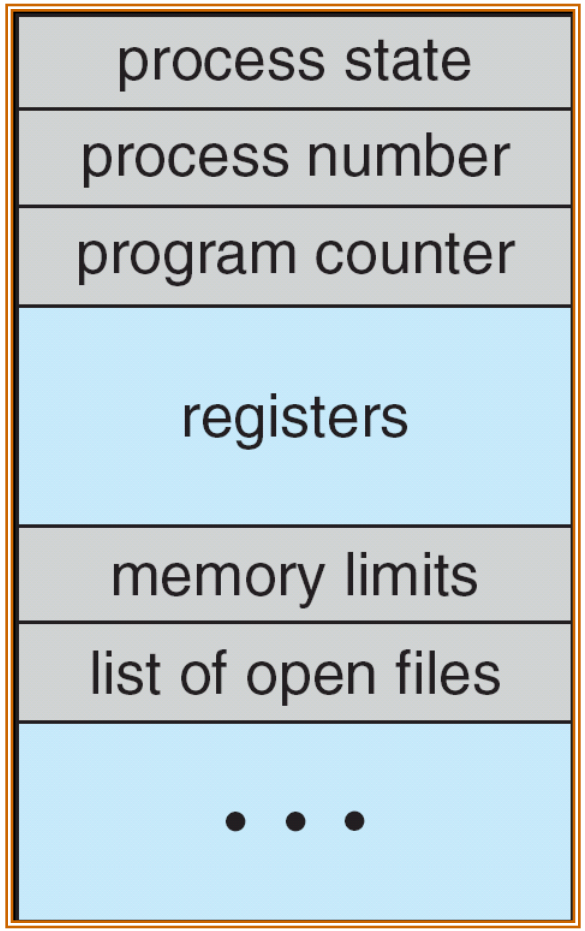
- **Running → Waiting**
  - When a process requests a service from the OS and the OS cannot satisfy immediately (software interruption due to a system call)
    - An access to a resource not yet available
    - Starts an I/O: must wait for the result
    - Needs a response from another process
- **Waiting → Ready**
  - When the expected event has occurred.
- **Running → Terminated**
  - The process has reached the end of the program (or an error occurred).


PCB
Program Execution
Process State
Process Switching
Concept
Scheduling
max
stack
Process
heap
data
text
0
Operations
Cooperation: IPC

Saving process information

- In a multitasking system, a process executes in an intermittent fashion on the CPU.
- Each time a process is assigned the CPU (transitions from ready to running), execution must continue in the same situation the process left the CPU last (i.e. same content in the CPU registers, etc.)
- Thus when a process leaves the running state, it is necessary to save important process information that will be retrieved when it comes back to the running state.

Process Control Block (PCB)



Process Control Block (PCB)

PCB Contains all information the OS needs to remember about a process

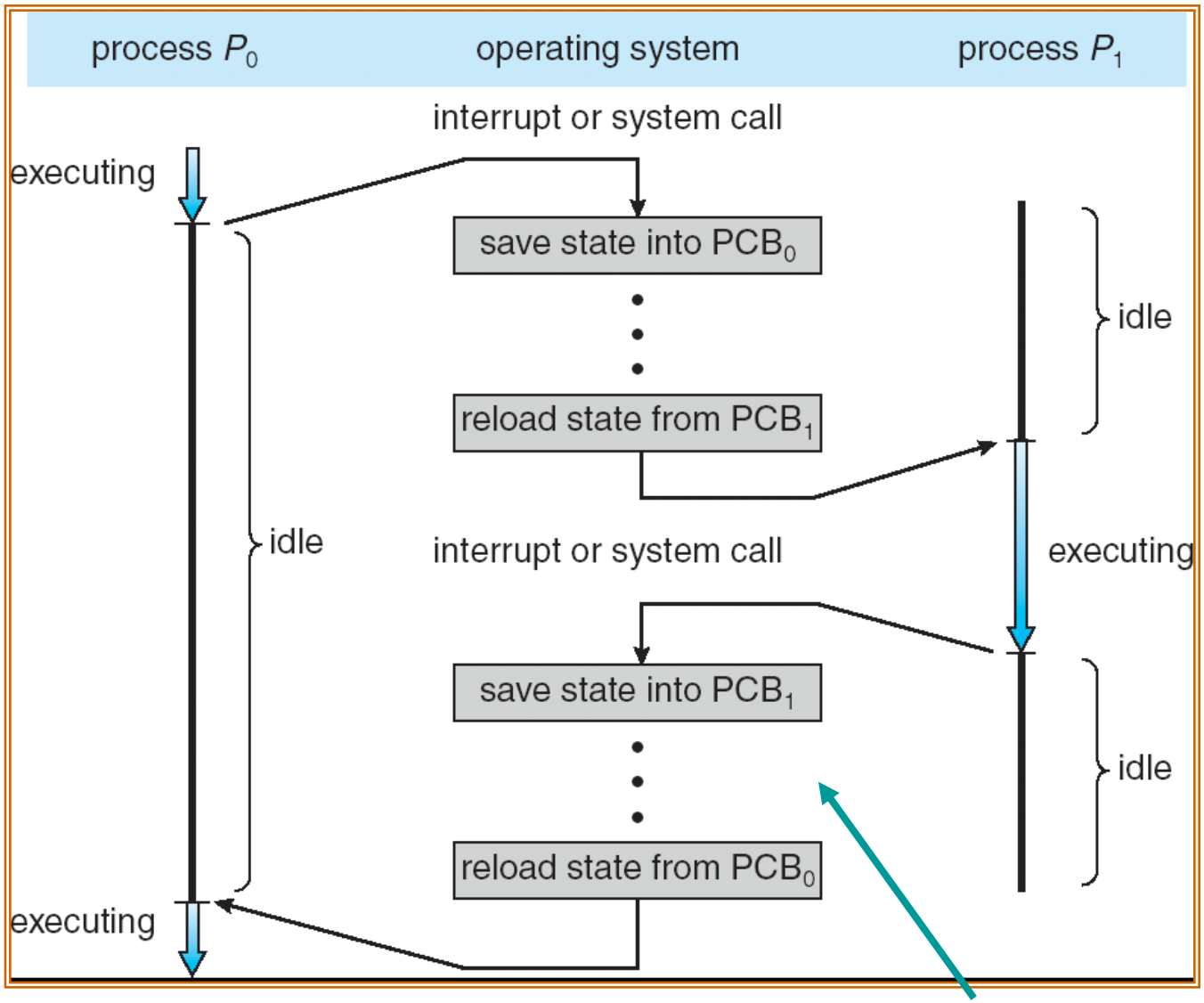
So, what is needed?

- To resume process execution?
  - Program counter
  - CPU registers
- To know how to schedule the process?
  - CPU scheduling information (i.e. priority)
- To know which memory addresses the process can access?
  - Memory-management information (base and limit pointers, …)
- What else?
  - Process state
  - I/O status information (e.g. open files)
  - Owner, parent process, children
  - Accounting information

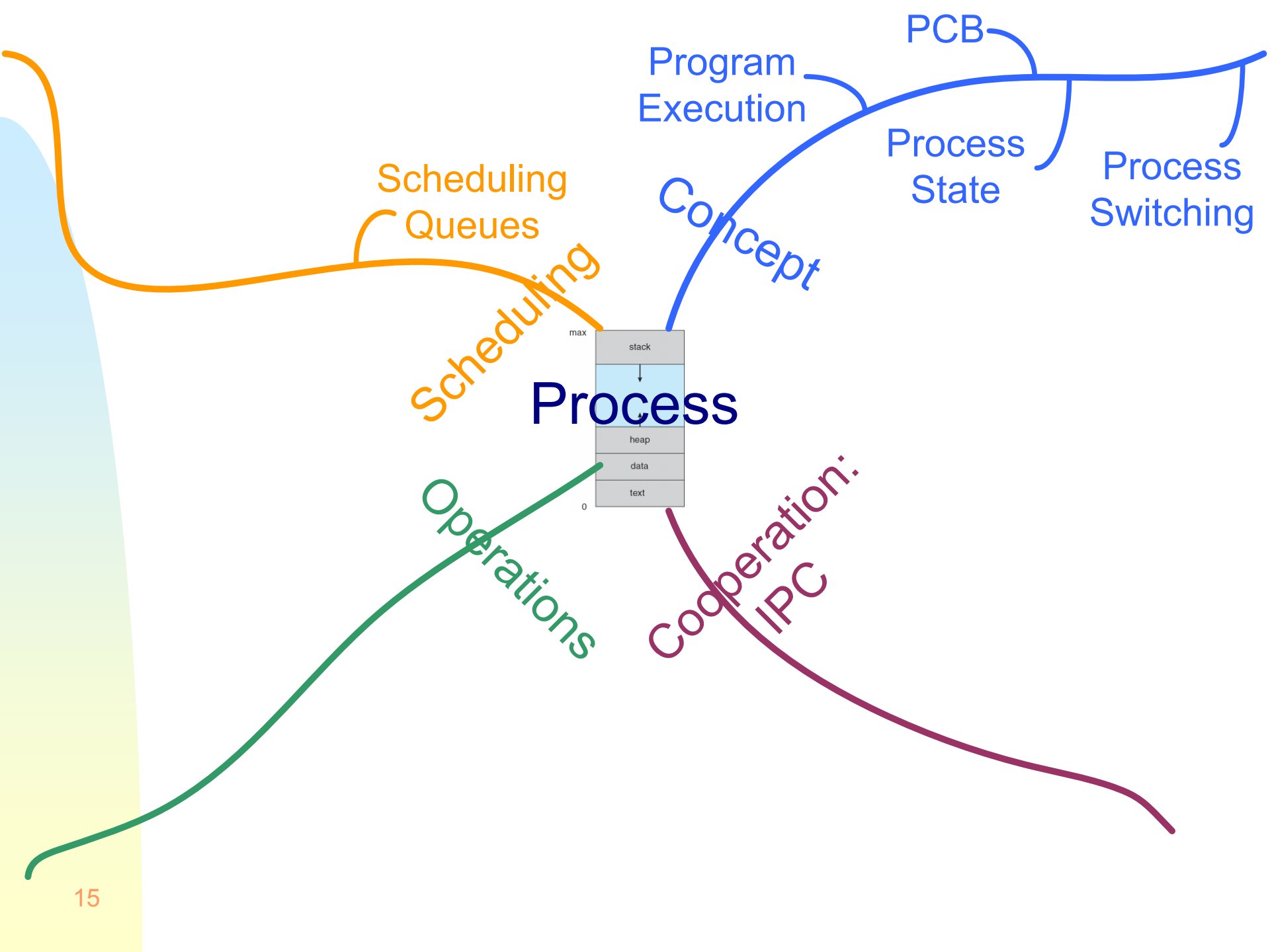
Process switching
Also called context switching

- When the CPU moves from executing of one process, say process 0, to another process, say process 1
  - The context switching occurs when the CPU is interrupted during the execution of process 0 (can be either hardware or software interrupt).
  - Updates and saves the PCB of process 0
  - Finds and accesses the PCB of process 1, that was previously saved.
  - Updates the CPU registers, program counter, etc. to restore the context as defined in PCB 1
  - Returns from the interrupt to allow the CPU to continue the execution of process 1.

CPU Switch From Process to Process



It is possible that much time passes before returning to Process 0, and that many other processes executes during this time


Process
Concept
Program Execution
PCB
Process State
Process Switching
Scheduling
Scheduling Queues
Operations
Cooperation: IPC
max
stack
heap
data
text
0

Process Scheduling

What?

- Choosing which process to run next

Why?

- Multiprocessing – to achieve good utilization
- Time sharing – to achieve low response time

How?

- We will talk in detail about that in Module 4 (Chapter 5).
- Now we just introduce the basic concepts of CPU scheduling.
- We have already seen the first one
  - Context switch from process to process

Process Scheduling Queues

What?

- **data structures supporting scheduling algorithms**

Job queue

- **set of all processes in the system**

Ready queue

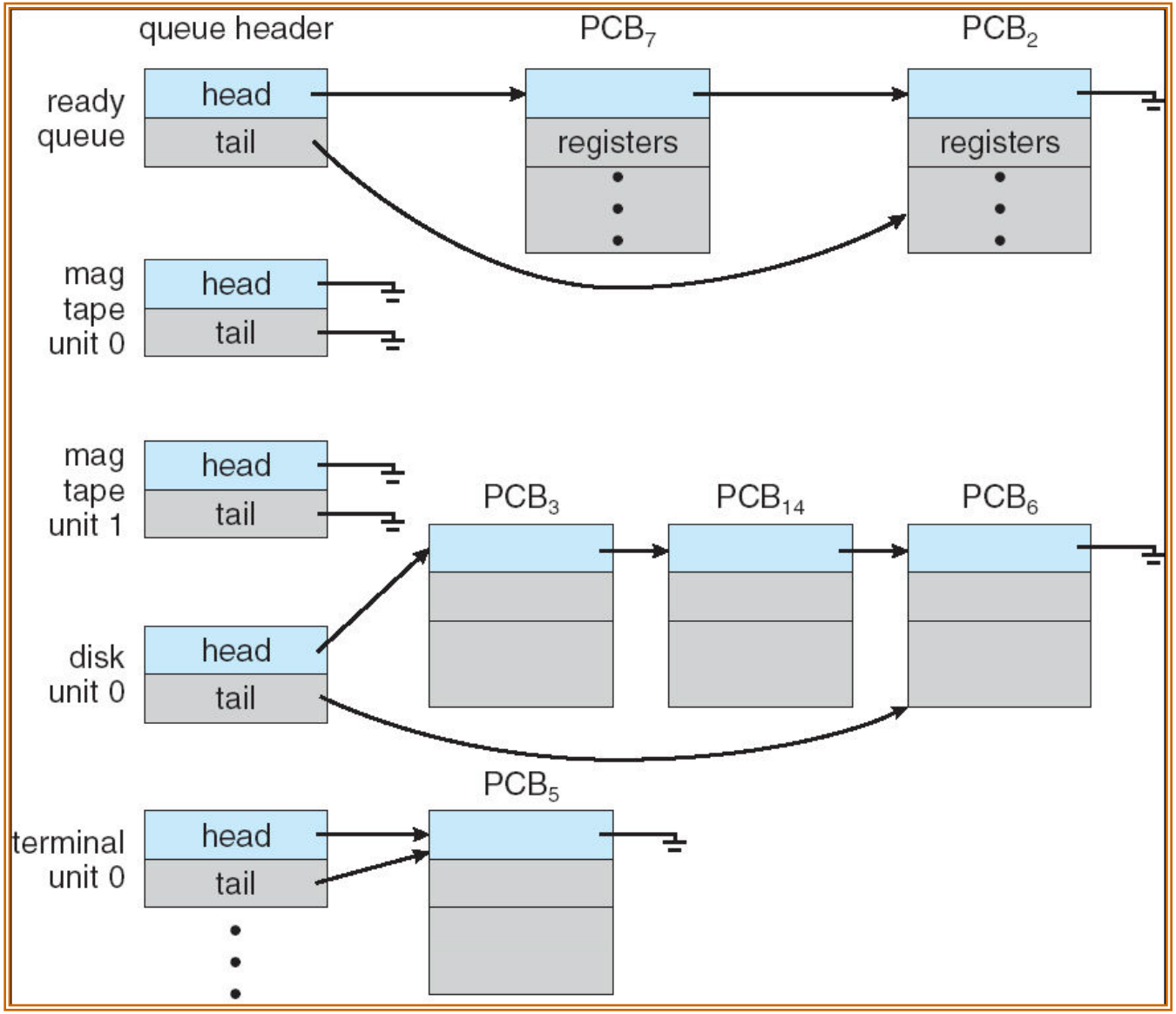
- **set of all processes residing in main memory, ready and waiting to execute**

Device queues

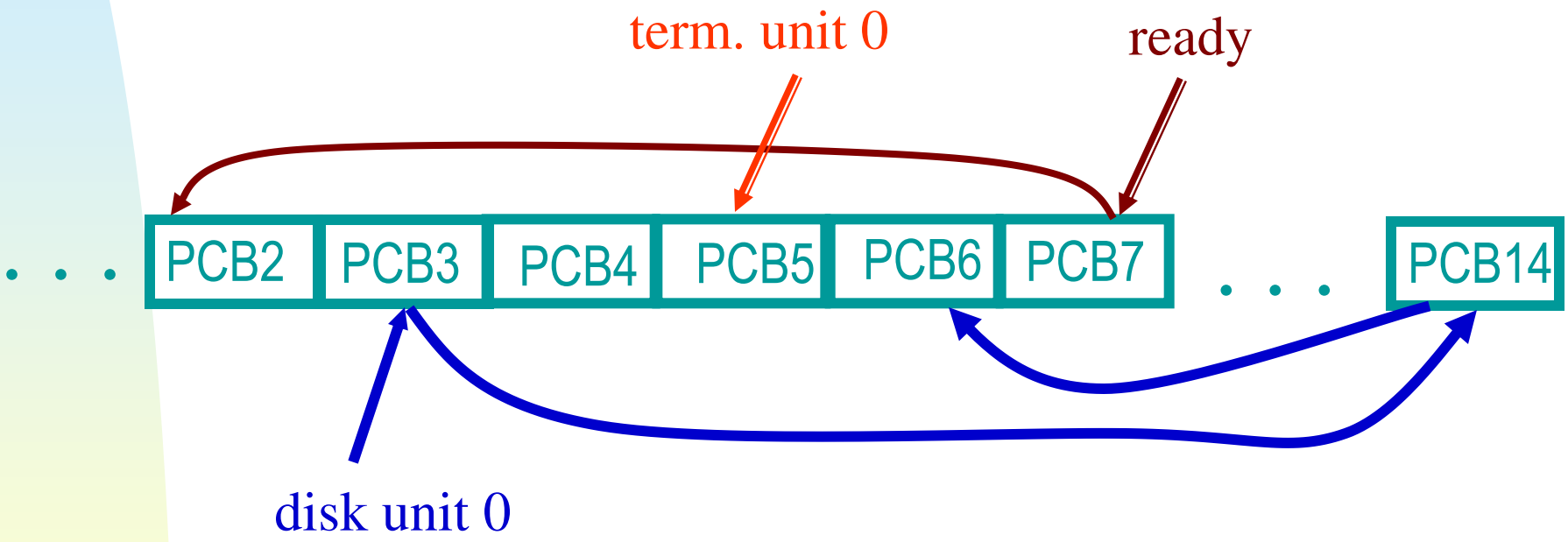
- **for I/O devices**
- **set of processes waiting for an I/O on that device**

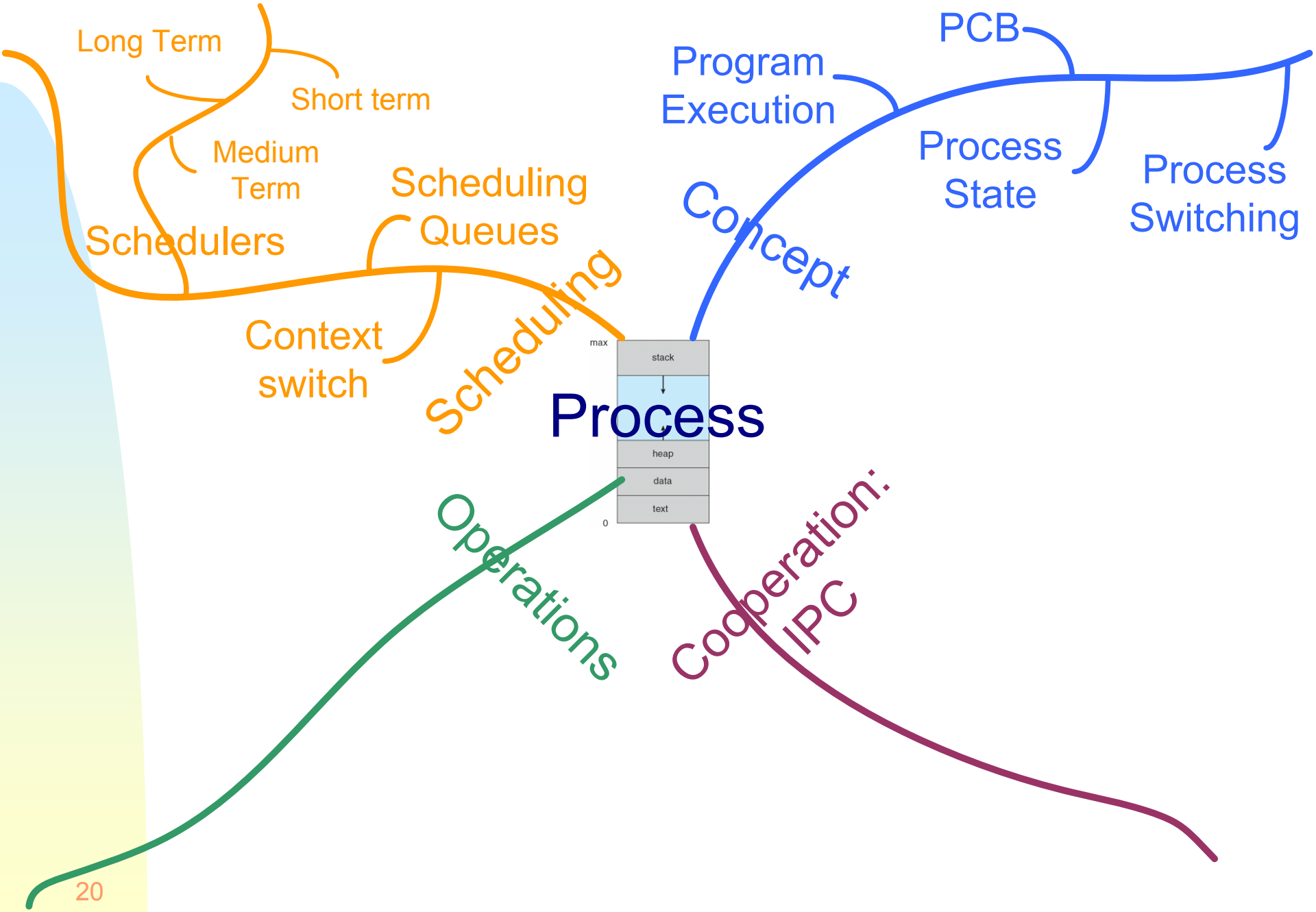
Processes (i.e. their PCB's) migrate between the various queues as they change state

Ready Queue And Various I/O Device Queues



The PCBs are not moved around in memory to be put into different queues: it is pointers in the PCBs that are used to create the queues.




Long Term
Short term
Medium Term
Schedulers
Scheduling Queues
Context switch
Scheduling
Concept
PCB
Program Execution
Process State
Process Switching
max
stack
heap
data
text
0
Process
Operations
Cooperation: IPC

Schedulers

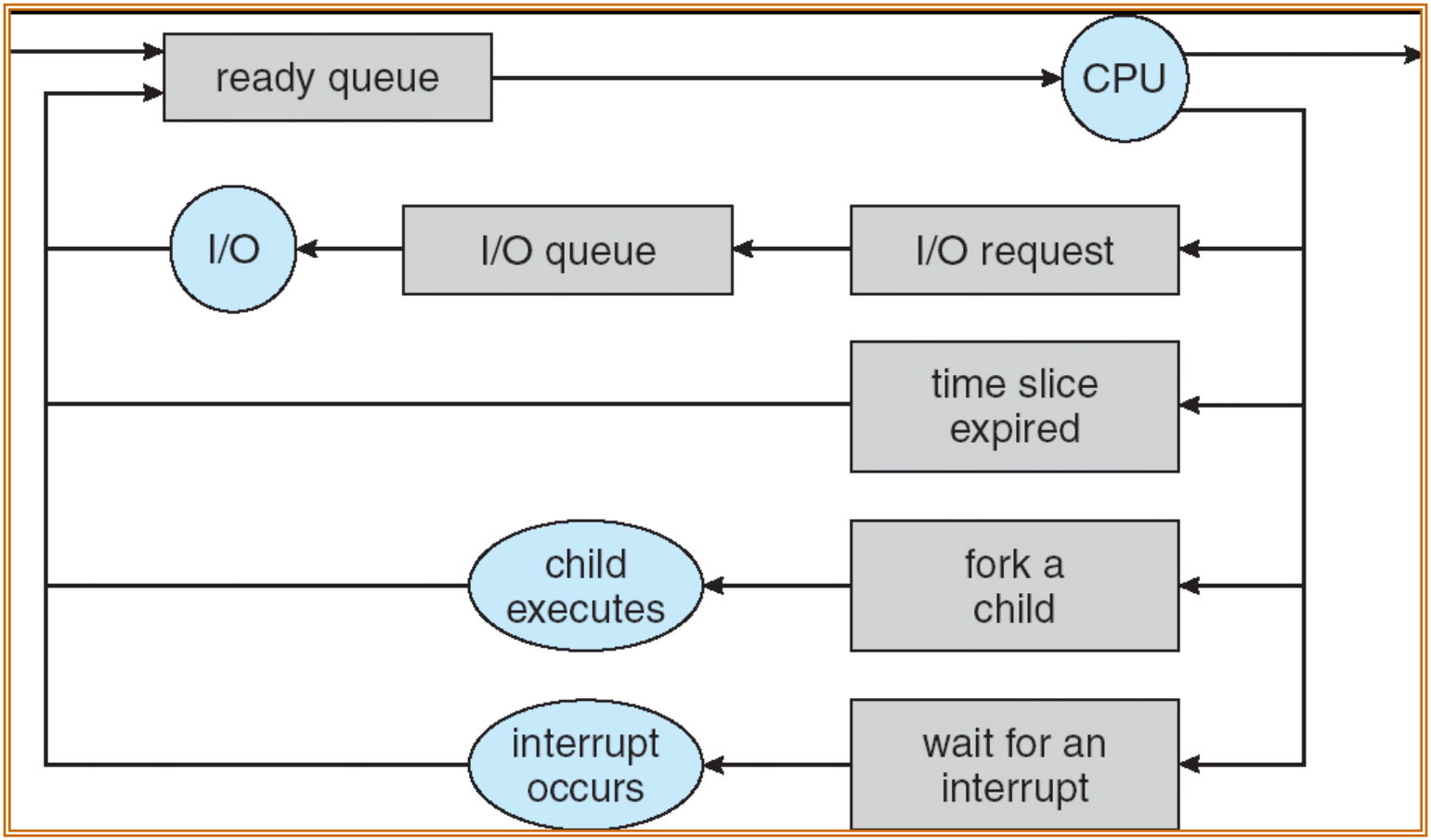
- Long-term scheduler **(or job scheduler)**
  - **selects which new processes should be brought into the memory; new → ready (and into the ready queue from a job spool queue) (used in batch systems)**
- Short-term scheduler **(or CPU scheduler)**
  - **selects which ready process should be executed next; ready → running**

- **Which of these schedulers must execute really fast, and which one can be slow? Why?**
- **Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)**
- **Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)**

Schedulers (Cont.)

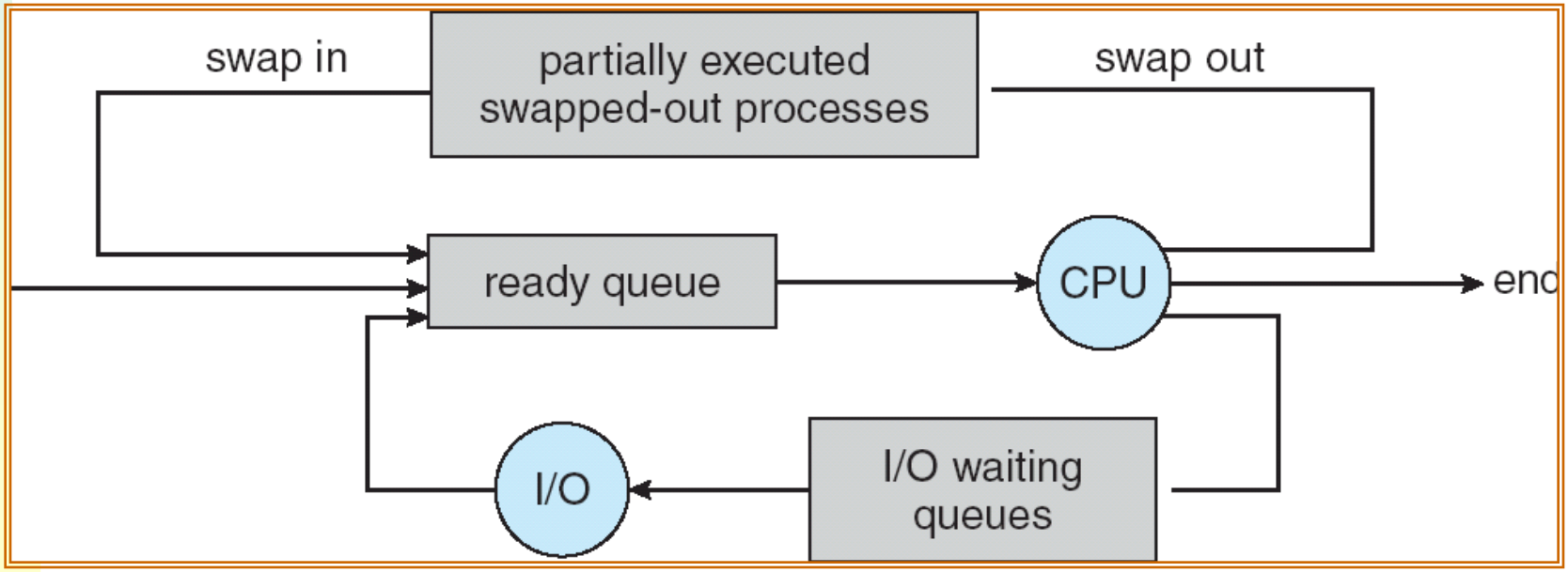
- **Processes differ in their resource utilization:**
  - I/O-bound process **– spends more time doing I/O than computations, many short CPU bursts**
  - CPU-bound process **– spends more time doing computations; few very long CPU bursts**
- **The long-term scheduler controls the *degree of multiprogramming***
  - **the goal is to efficiently use the computer resources**
  - **ideally, chooses a mix of I/O bound and CPU-bound processes**
    - **but difficult to know beforehand**

Representation of Process Scheduling



Medium Term Scheduling

- **Due to memory shortage, the OS might decide to swap-out a process to disk.**
  - **Later, it might decide to swap it back into memory when resources become available**
- Medium-term scheduler **– selects which process should be swapped out/in**



Context Switch

What?

- **Assigning the CPU from one process to another**

How?

- **We have already seen that**
- **Save the CPU state to the PCB of the current process**
- **Load the CPU with the information from the PCB of the new process**
- **Set the program counter to the PC of the new process**
- **Additional work as well (accounting, …)**

Comments:

- **Can be quite a lot of work - pure overhead, as no process is executing at that time**
- **Context switch time is dependent on hardware support (and OS smarts)**

Long Term

Short term

Medium Term

Schedulers

Scheduling Queues

Context switch

Scheduling

createProcess

fork/exec

Process Creation

Operations

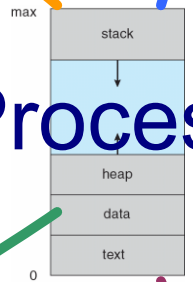
Program Execution

PCB

Process State

Process Switching

Concept



Process

Cooperation: IPC

Process Creation

So, where do all processes come from?

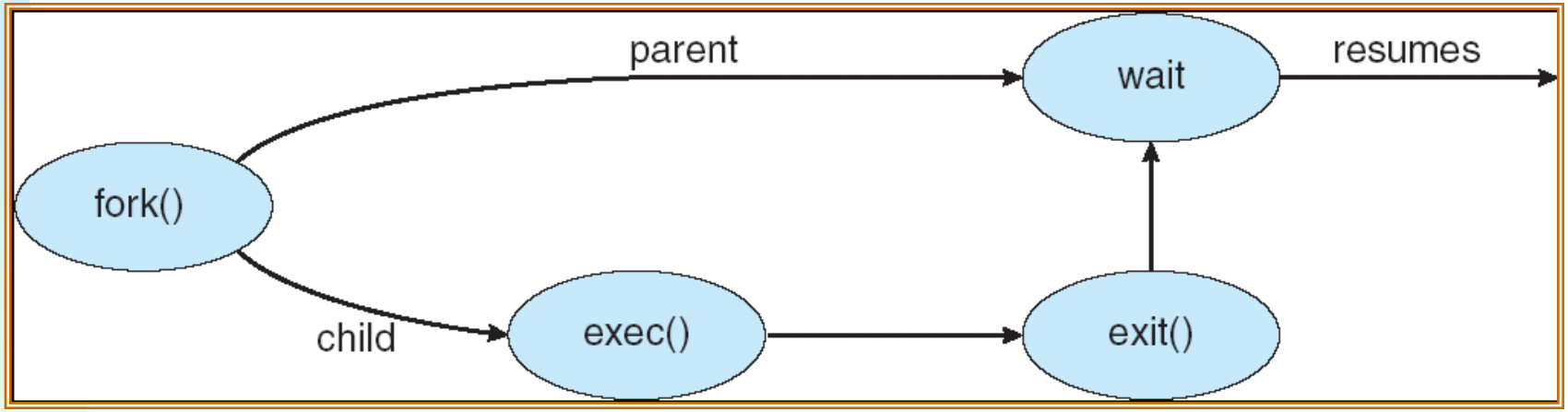
- **Parent process create children processes, which, in turn create other processes, forming a tree of processes**
- **Usually, several properties can be specified at child creation time:**
  - **How do the parent and child share resources?**
    - Share all resources
    - Share subset of parent's resources
    - No sharing
  - **Does the parent run concurrently with the child?**
    - Yes, execute concurrently
    - No, parent waits until the child terminates
  - **Address space**
    - Child duplicate of parent
    - Child has a program loaded into it

Process Creation (Cont.)

UNIX example:

- fork() **system call creates new process with the duplicate address space of the parent**
  - **no shared memory, but a copy**
  - **copy-on-write used to avoid excessive cost**
  - **returns child's pid to the parent, 0 to the new child process**
  - **the parent may call** wait() **to wait until the child terminates**
- exec(…) **system call used after a** fork() **to replace the process' memory space with a new program**

UNIX: fork(), exec(), exit() & wait()



C Program Forking Separate Process

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
  int pid;
   pid = fork();     /* fork another process */
   if (pid < 0) {      /* error occurred */
          fprintf(stderr, "Fork Failed");
          exit(-1);
   } else if (pid == 0) { /* child process */
          execlp("/bin/ls","ls",NULL);
   } else { /* parent process, will wait for the
                child to complete */
          wait(NULL);
          printf("Child Complete");
          exit(0);
   }
}
```

Fork example

```
int pid, a = 2, b=4;
pid = fork();    /* fork another process */
if (pid < 0)  exit(-1); /* fork failed */
else if (pid == 0) { /* child process */
    a = 3; printf("%d\n", a+b);
} else {
    wait();
    b = 1;
    printf("%d\n", a+b);
}
```

What would be the output printed?

7

3

Fork example

```
int pid, a = 2, b=4;
pid = fork();     /* fork another process */
if (pid < 0)  exit(-1); /* fork failed */
else if (pid == 0) { /* child process */
    a = 3; printf("%d\n", a+b);
} else {
    b = 1;
    wait();
    printf("%d\n", a+b);
}
```

What would be the output printed?

7

3

Fork example

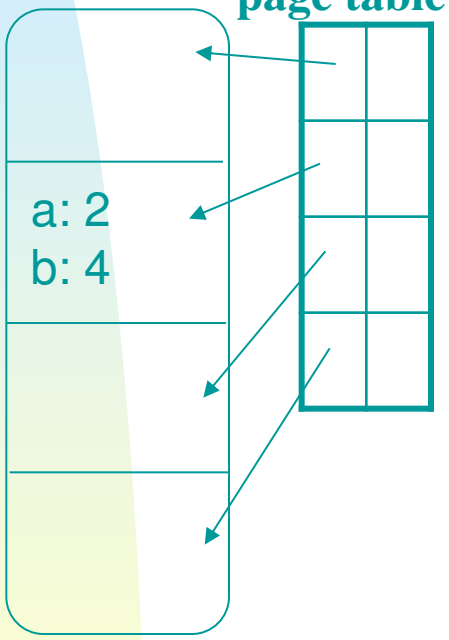
```
int pid, a = 2, b=4;
pid = fork();   /* fork another process */
if (pid < 0)  exit(-1); /* fork failed */
else if (pid == 0) { /* child process */
    a = 3; printf("%d\n", a+b);
} else {
    b = 1;
    printf("%d\n", a+b);
    wait();
}
```

What would be the output printed?

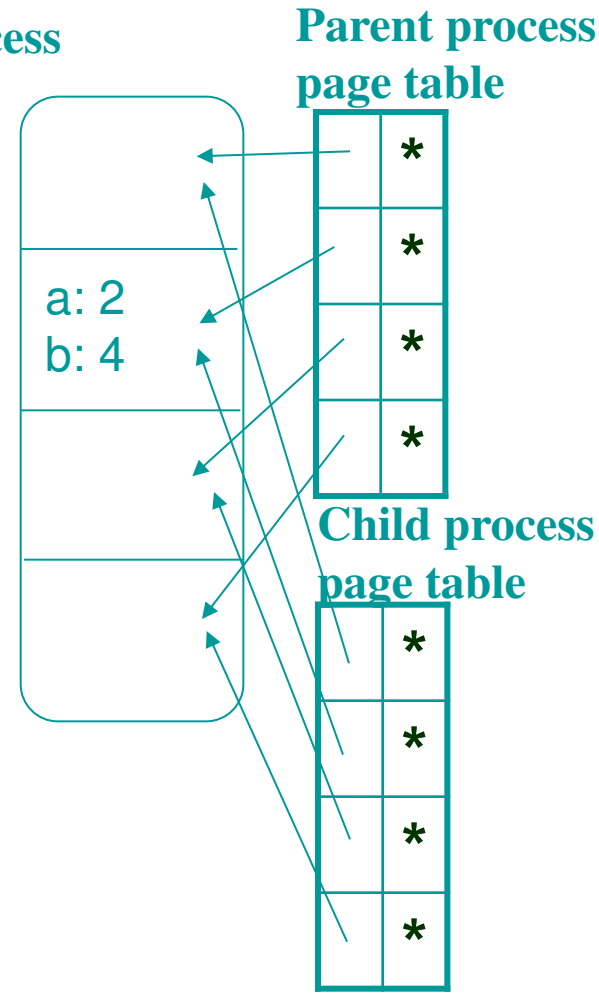
7	or	3
3		7

Understanding fork()

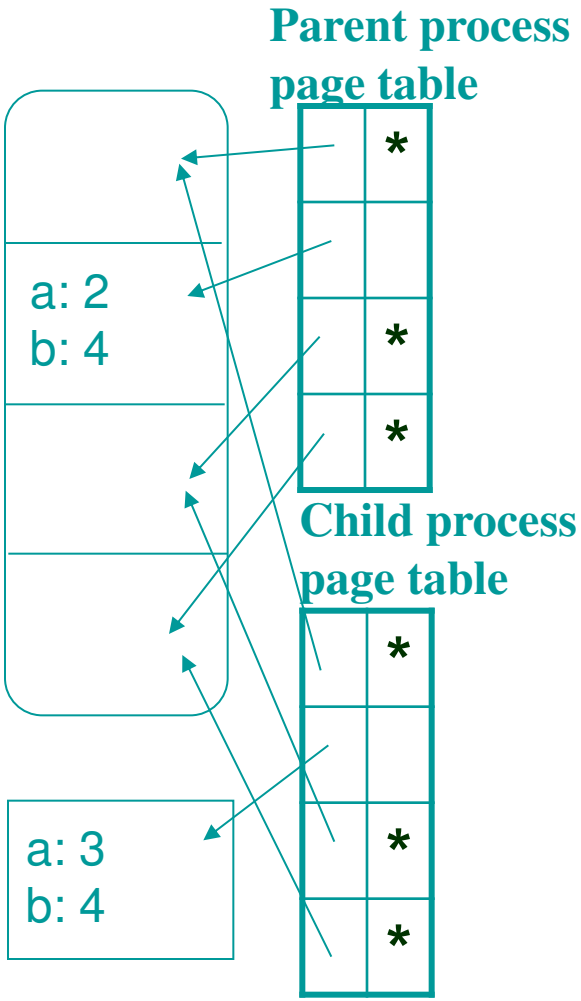
Before fork



After fork



After child executes a=3



Process Creation (Cont.)

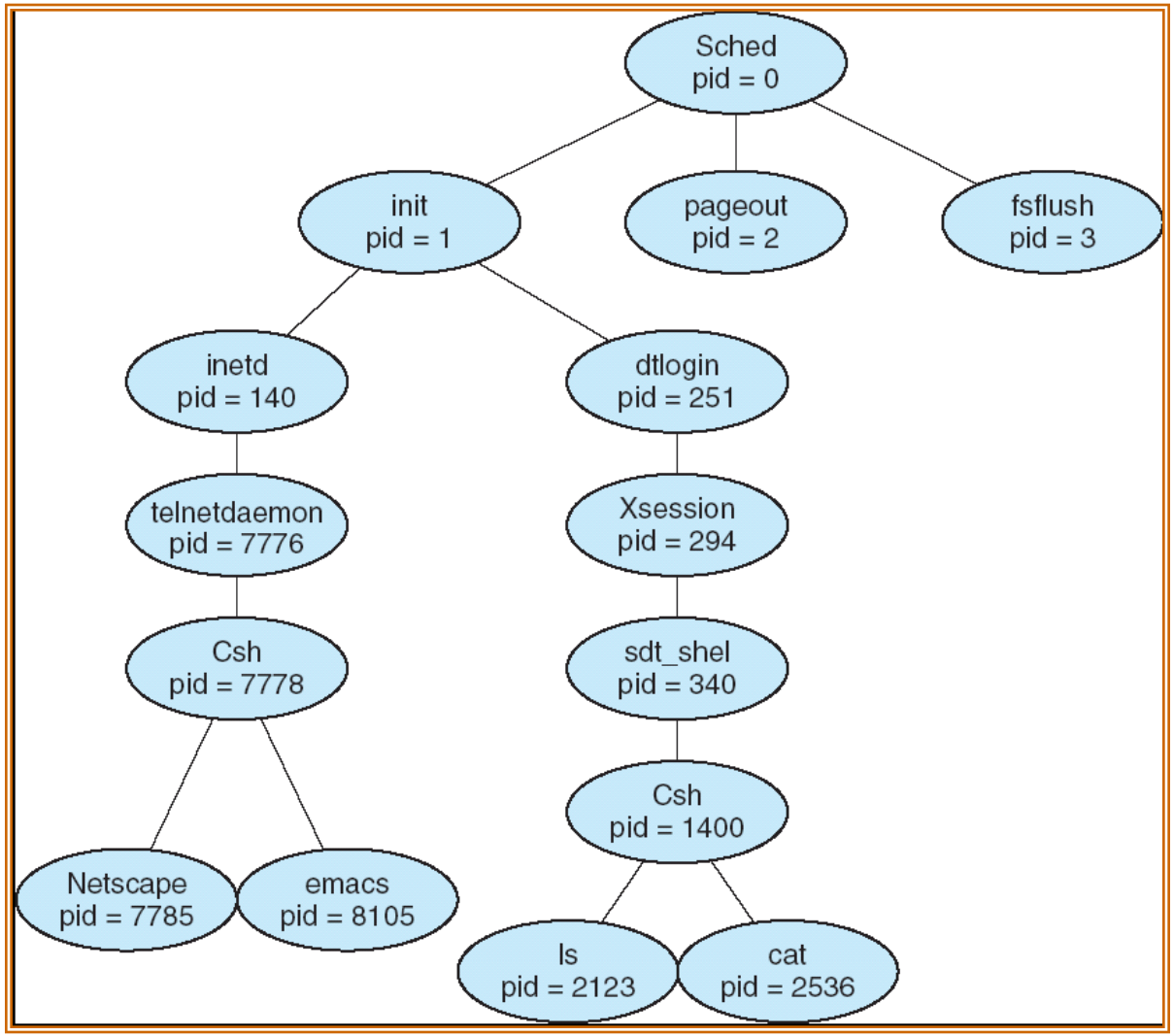
Windows:

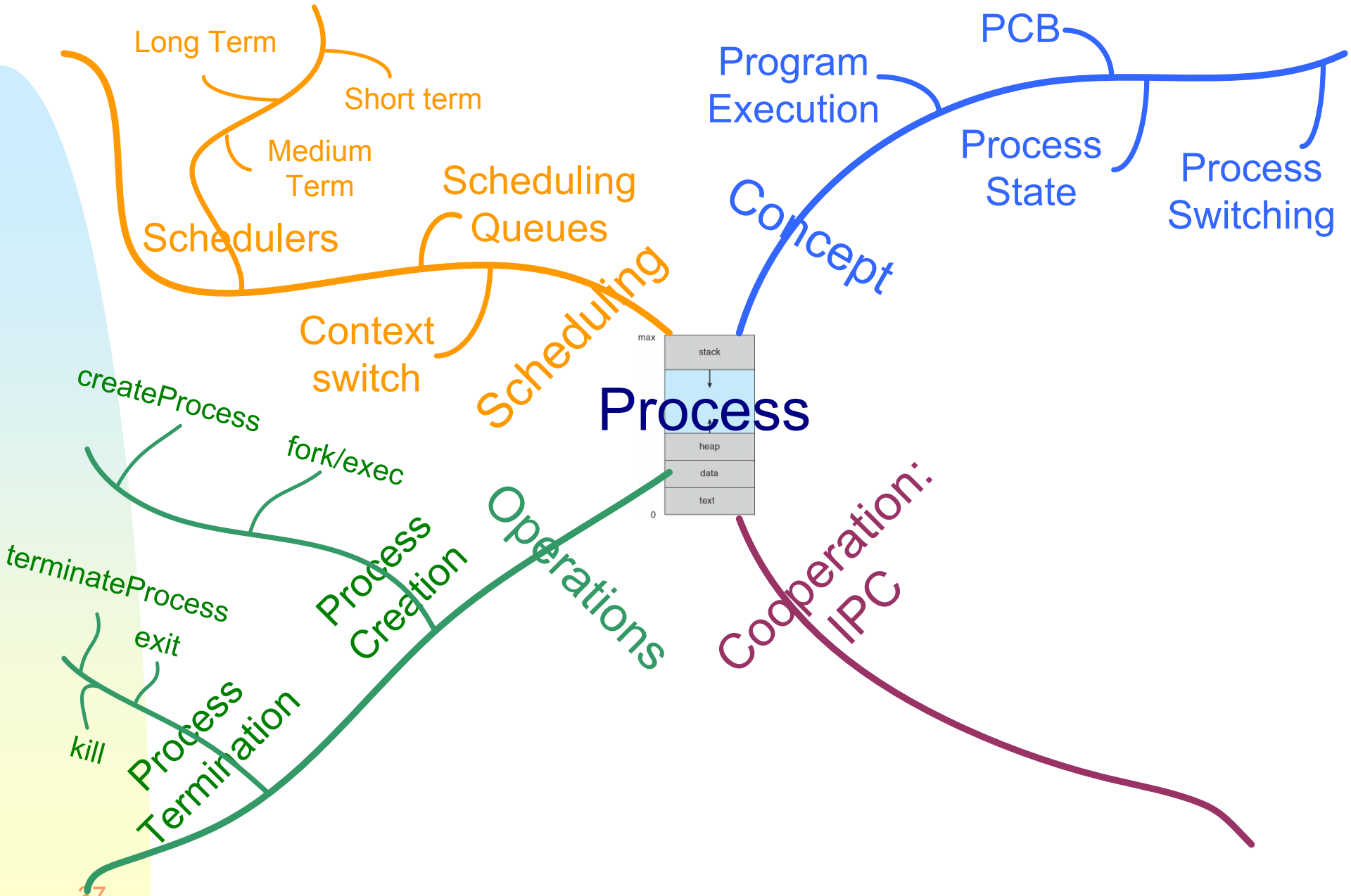
- CreateProcess(…)
- **Similar to fork() immediately followed by exec()**

Discussion

- **fork() very convenient for passing data/parameters from the parent to the child**
- **all code can be conveniently at one place**
- **direct process creation more efficient when you really just want to launch a new program**

Example Process Tree (Solaris)




Long Term
Short term
Medium Term
Schedulers
Scheduling Queues
Context switch
Scheduling
Program Execution
PCB
Process State
Process Switching
Concept
max
stack
heap
data
text
0
Process
createProcess
fork/exec
Process Creation
Operations
terminateProcess
exit
kill
Process Termination
Cooperation: IPC

Process Termination

How do processes terminate?

- **Process executes last statement and asks the operating system to delete it (by making** exit() **system call)**
- **Abnormal termination**
  - **Division by zero, memory access violation, …**
- **Another process asks the OS to terminate it**
  - **Usually only a parent might terminate its children**
    - **To prevent user's terminating each other's processes**
  - **Windows: TerminateProcess(…)**
  - **UNIX: kill(processID, signal)**

Process Termination

What should the OS do?

- **Release resources held by the process**
  - **When a process has terminated, but not all of its resources has been released, it is in state** terminated **(**zombie**)**
- **Process' exit state might be sent to its parent**
  - **The parent indicates interest by executing** wait() **system call**

What to do when a process having children is exiting?

- **Some OSs (VMS) do not allow children to continue**
  - **All children terminated - *cascading termination***
- **Other find a parent process for the orphan processes**

Time for questions

Process states

- **Can a process move from waiting state to running state?**
- **From ready state to terminated state?**

PCB

- **Does PCB contain program's global variables?**
- **How is the PCB used in context switches?**

CPU scheduling

- **What is the difference between long term and medium term scheduler?**
- **A process (in Unix) has executed wait() system call. In which queue it is located?**
- **What happens if there is no process in the ready queue?**

Other questions!

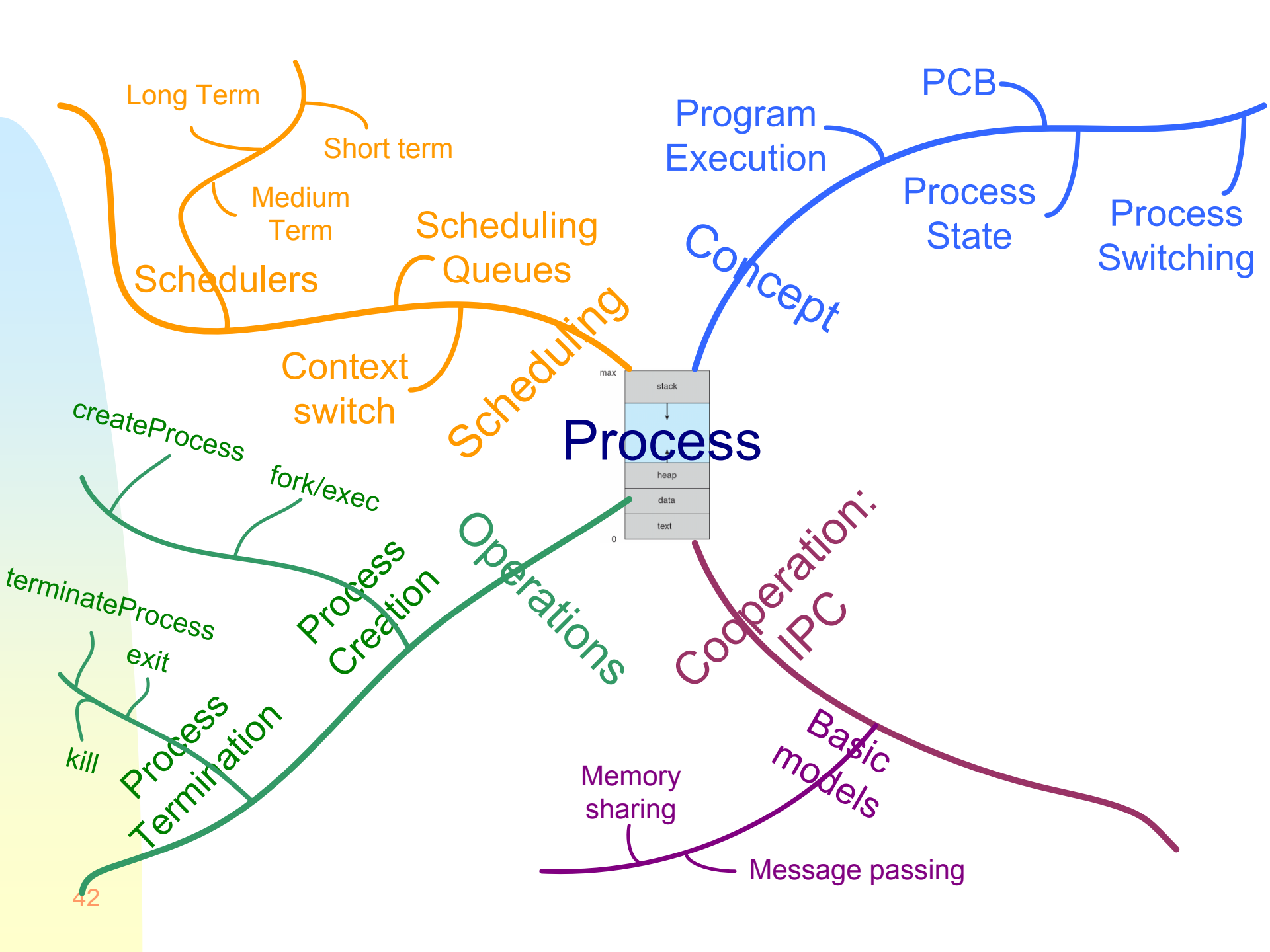
Process creation

- **Understand fork(), exec(), wait()**
- **Do you think you understand them?**
  - **So, how many processes are created in this code fragment?**

```
for(i=0; i<3; i++)
  fork();
```

Process termination

- **How/when.**
- **What should the OS do?**


Process
Scheduling
Long Term
Short term
Medium Term
Schedulers
Scheduling Queues
Context switch
Concept
Program Execution
PCB
Process State
Process Switching
max
stack
heap
data
text
0
Operations
createProcess
fork/exec
Process Creation
terminateProcess
exit
kill
Process Termination
Cooperation: IPC
Basic models
Memory sharing
Message passing

Cooperating Processes

- *Independent* processes cannot affect or be affected by the execution of other processes
- *Cooperating* processes can affect or be affected by the execution of other processes
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up (parallel processing)
  - Modularity
  - Nature of the problem may request it
  - Convenience

So, we really want Inter-Process Communication (IPC)

Interprocess Communication (IPC)

- Mechanisms for processes to communicate and to synchronize their actions
- So, how can processes communicate?
  - Fundamental models of IPC
    - Through shared memory
    - Using message passing
  - Examples of IPC mechanisms
    - signals
    - pipes & sockets
    - semaphores …

Shared Memory

- By default, address spaces of processes are disjoint
  - To protect from unwanted interference
- The OS has to allow one process to access (a precisely specified) part of the address space of another process
  - One process makes a system call to create a shared memory region
  - Other process makes system call to map this shared memory region into its address space
- Special precaution is needed in accessing the data in the shared memory, it is way too easy to get into inconsistent state
  - We will talk about this extensively in Chapter 6

Shared Memory - Server:

```
...
#define SHMSZ     27
 main()    {
        char c;
        int shmid;
        key_t key;
        char *shm, *s;

        key = 5678;
        if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
             perror("shmget");  exit(1);  }

       if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
             perror("shmat"); exit(1);  }

       s = shm;
       for (c = 'a'; c <= 'z'; c++)    *s++ = c;
        *s = '\0';

        while (*shm != '*')  sleep(1);
        exit(0);
}
```

Shared Memory - Client

```
…
#define SHMSZ     27
main(){
        int shmid;
        key_t key;
        char *shm, *s;

         key = 5678;
         if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
                 perror("shmget");
                 exit(1);
          }

          if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                   perror("shmat");  exit(1);  }

          for (s = shm; *s != '\0'; s++)  putchar(*s);
          putchar('\n');

           *shm = '*';

           exit(0);
}
```

Message Passing

- **Two basic operations:**
  - send(**destination, message**)
  - receive(**source, message**)
- **If processes wish to communicate, they need to establish a *communication link* between them**
  - **figure out what destination and source to use**
- **There are many variants of how send() and receive() behave**
  - **Direct or indirect communication**
  - **Synchronous or asynchronous communication**
  - **Automatic or explicit buffering**

Direct Communication

- **Processes must name each other explicitly:**
  - send (***P, message***) – **send a message to process P**
  - receive(***Q, message***) – **receive a message from process Q**
- **Properties of the communication link**
  - **Links are established automatically, exactly one link for each pair of communicating processes**
  - **The link may be unidirectional, but is usually bi-directional**

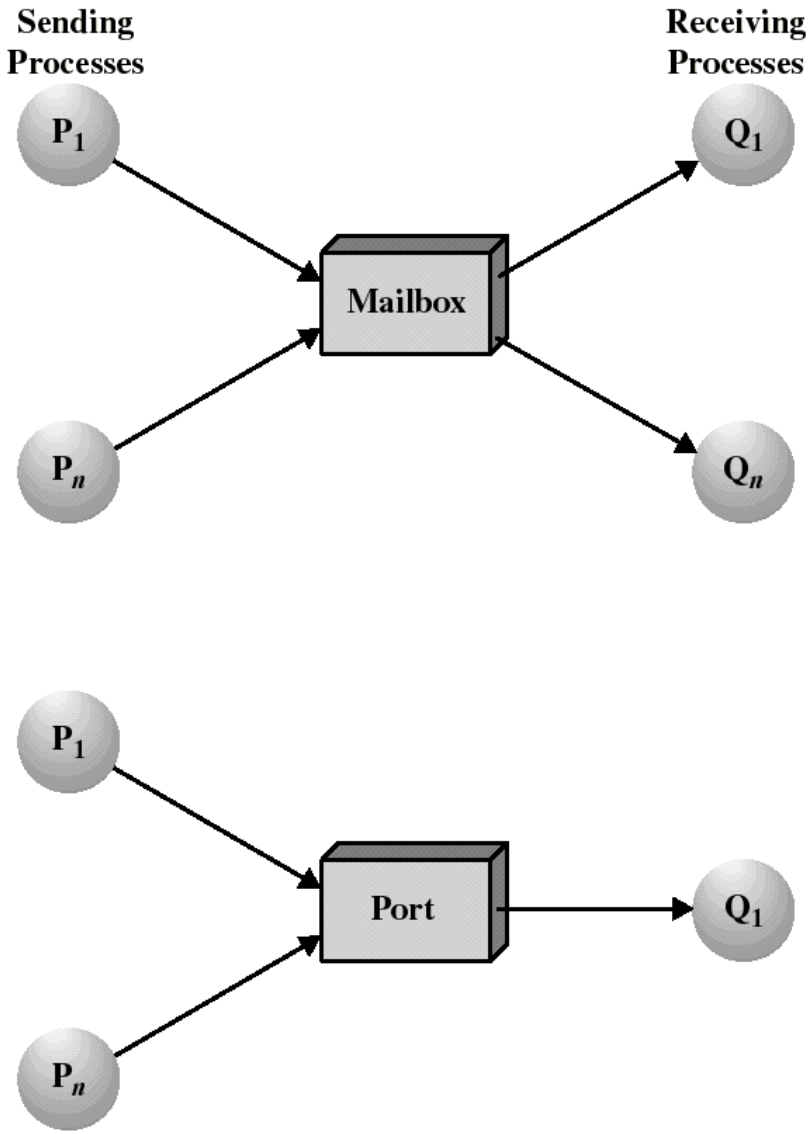
Indirect Communication

- **Messages are sent and received to/from mailboxes (also referred to as ports)**
  - **Each mailbox has unique id**
  - **Processes can communicate only if they share a mailbox**
- **Basic primitives:**

  send(***A, message***) **– send a message to mailbox A**

  receive(***A, message***) **– receive a message from mailbox A**
- **Operations**
  - **create a new mailbox**
  - **send and receive messages through mailbox**
  - **destroy a mailbox**

Indirect Communication



Indirect Communication

- **Mailbox sharing**
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- **Solutions**
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

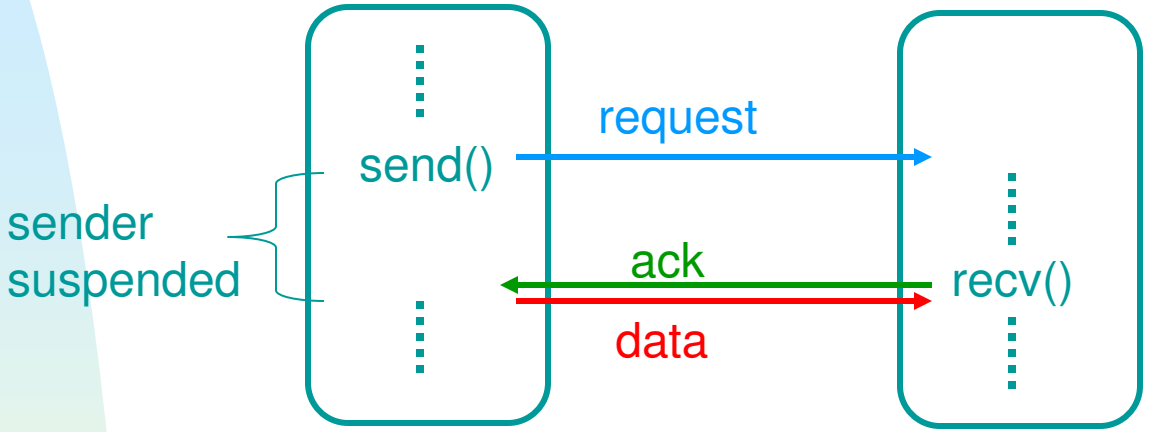
Blocking Message Passing

Also called synchronous **message passing**

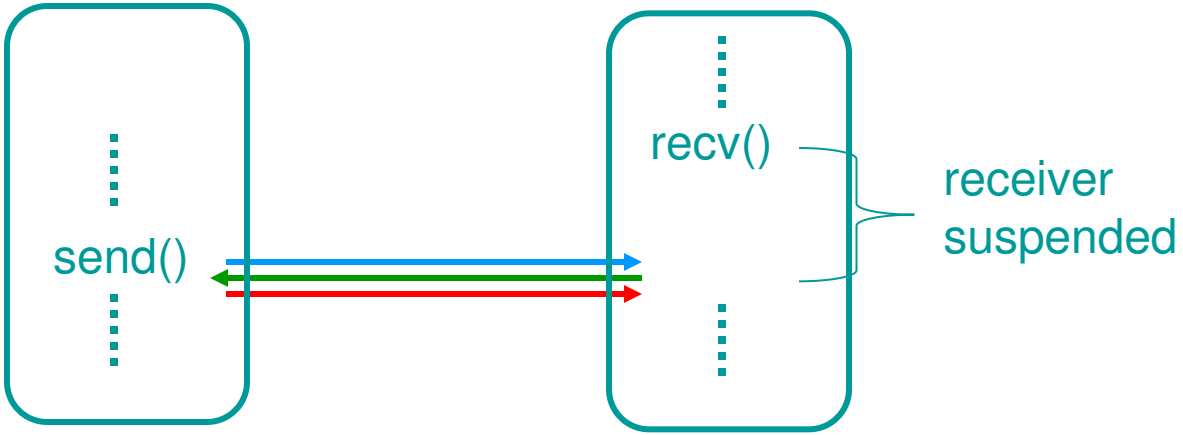
- **sender waits until the receiver receives the message**
- **receiver waits until the sender sends the message**
- advantages:
  - **inherently synchronizes the sender with the receiver**
  - **single copying sufficient**
- disadvantages:
  - **possible deadlock problem**

Synchronous Message Passing

send() before corresponding **receive()**



receive() before corresponding **send()**



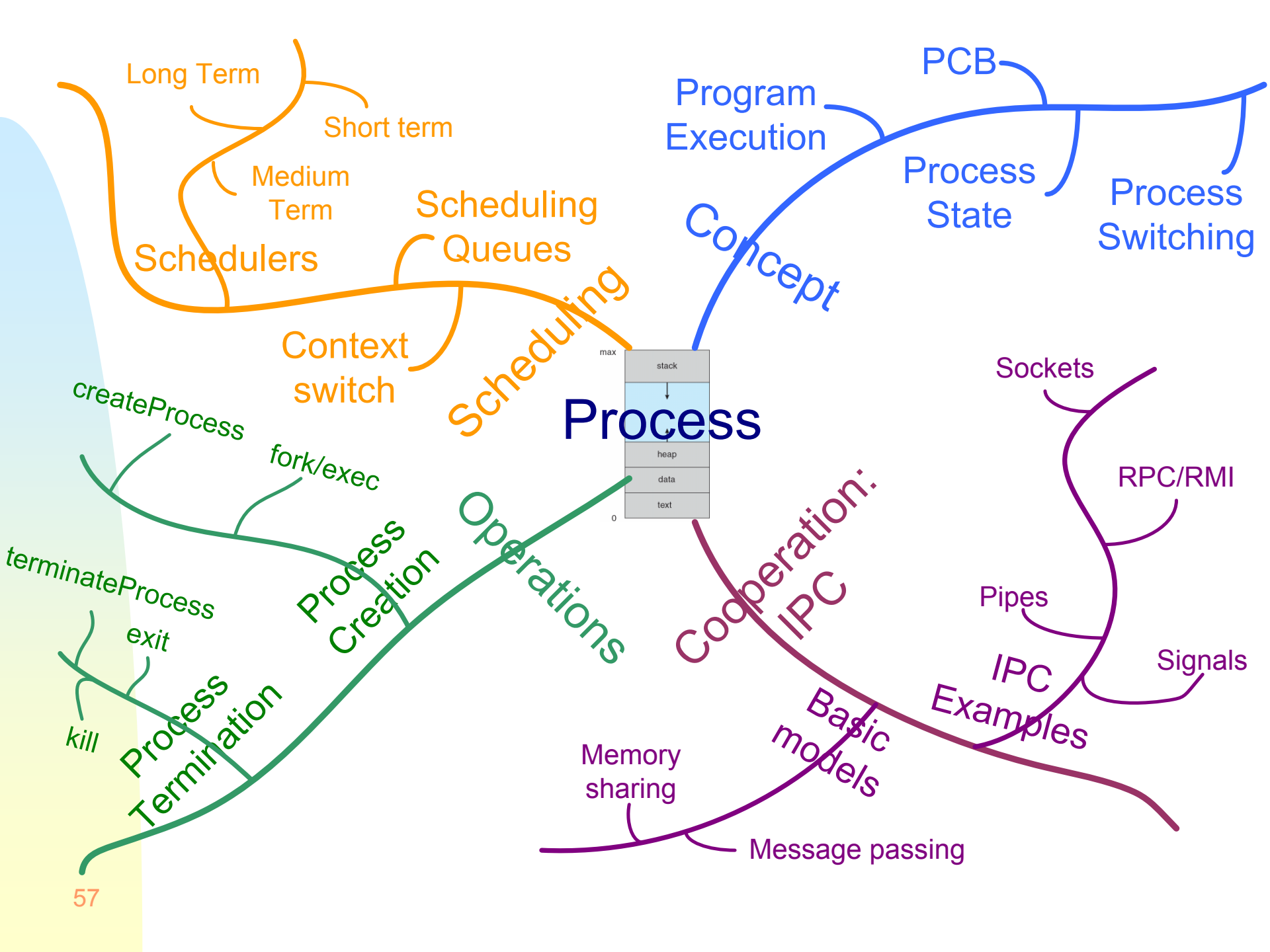
Non-Blocking Message Passing

Also called asynchronous **message passing**

- Non-blocking send: **the sender continues before the delivery of the message**
- Non-blocking receive: **check whether there is a message available, return immediately**

Buffering

- With blocking direct communication, no buffering is needed – the message stays in the sender's buffer until it is copied into the receiver's buffer
- With non-blocking communication, the sender might reuse the buffer,
  - therefore the message is copied to a system buffer,
  - and from there to the receiver's buffer
  - If the system buffer becomes full, the sender would still be suspended


Process
Scheduling
Long Term
Short term
Medium Term
Schedulers
Scheduling Queues
Context switch
Concept
Program Execution
PCB
Process State
Process Switching
max
stack
heap
data
text
0
Operations
createProcess
fork/exec
Process Creation
terminateProcess
exit
kill
Process Termination
Cooperation: IPC
Sockets
RPC/RMI
Pipes
Signals
IPC Examples
Basic models
Memory sharing
Message passing

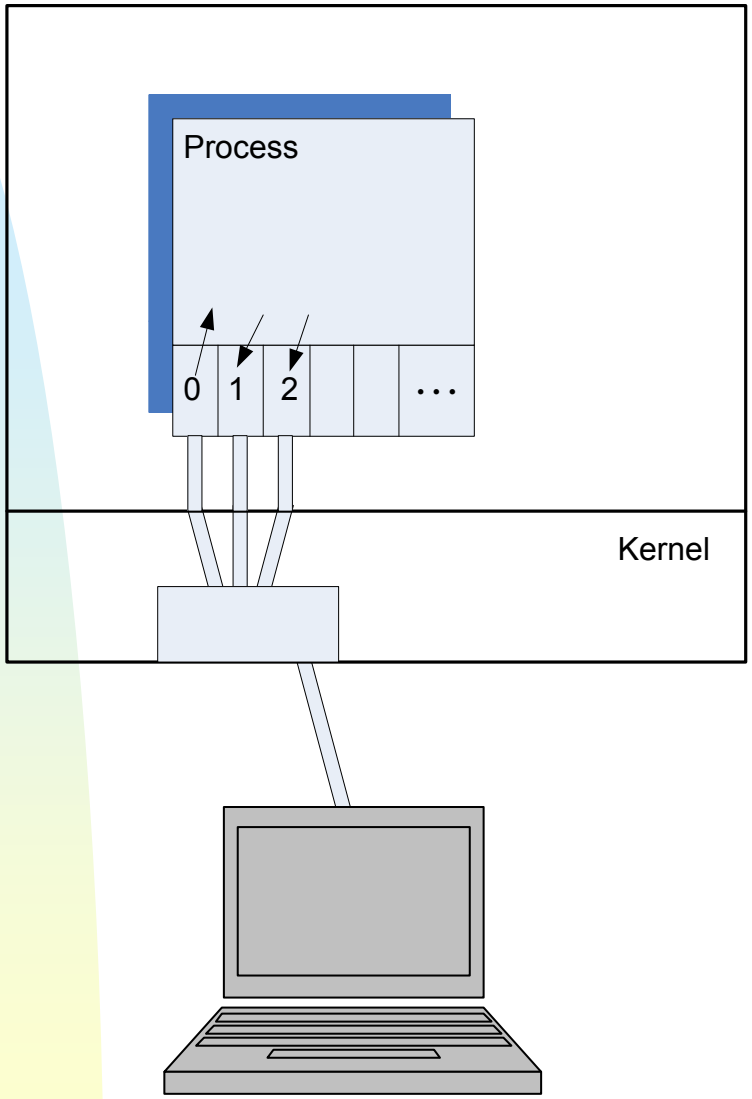
Examples of IPC Mechanisms

- Pipes
- Signals

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

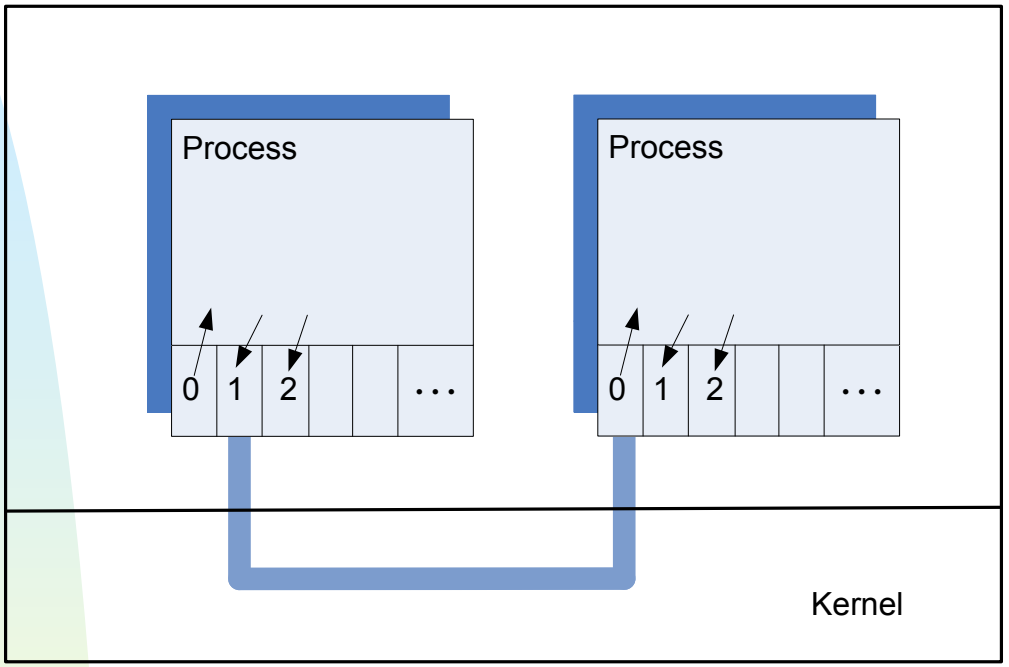
- Higher-level concepts
  - Implemented either using shared memory (between processes on the same computer) or message passing (between different computers)

Pipes – a few facts



- Each UNIX/Linux process is created with 3 open files:
  - 0: standard input
  - 1: standard output
  - 2: standard error
- Often attached to a terminal
- Can redirect to a file
  - cmd >file

Pipes - communication

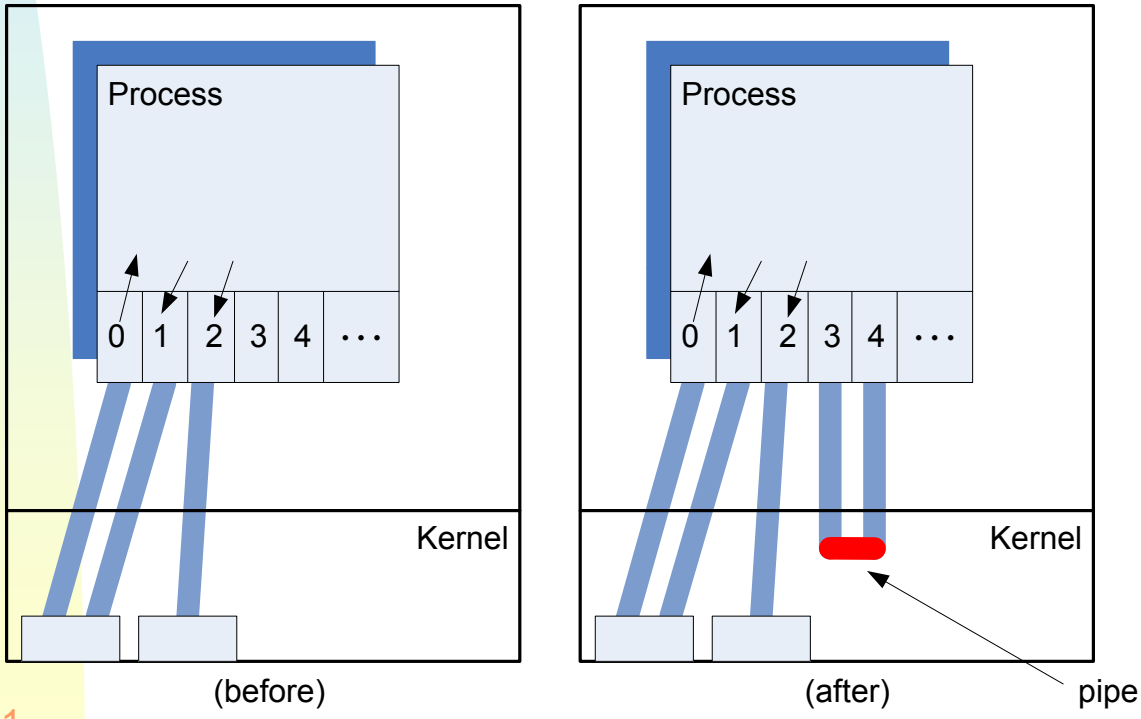


- **What is it?**
  - A unidirectional channel
  - One write end
  - One read end

- **Why?**
  - The UNIX/Linux pipes allows linking of processes to obtain complex functions from simple commands
  - who | sort

Pipes – how?

- System call: pipe(int *fd)
  - Create a pipe with two file descriptors:
  - fd[0] – reference to the read end
  - fd[1] - reference to the write end.




```
/* Array for storing 2 file
   descriptors */
int fd[2];
/* creation of a pipe */
int ret = pipe(fd);
if (ret == -1)  { /* error */
   perror("pipe");
   exit(1);
}
```

Unix Pipes

So, what did we get?

- fd[0] is a file descriptor for the read end of the pipe
  - read(fd[0], dataBuf, count) would read count bytes or less from the pipe into the dataBuf character array
- fd[1] is a file descriptor for the write end of the pipe
  - write(fd[1], dataBuf, count) would write count bytes from the dataBuf character array into the pipe

- Nice, but we want to communicate between different processes, and pipe() gives both endpoints to the process calling pipe()
  - Too bad, we are allowed to talk only with ourselves :-(
  - Wait! Can we talk at least within the family?

Unix Pipes

Note:

- When parent calls fork(), the child gets the parent's open file descriptors.
- That includes the pipe endpoints!
- If the child calls exec() to run another program, the new open files are preserved.

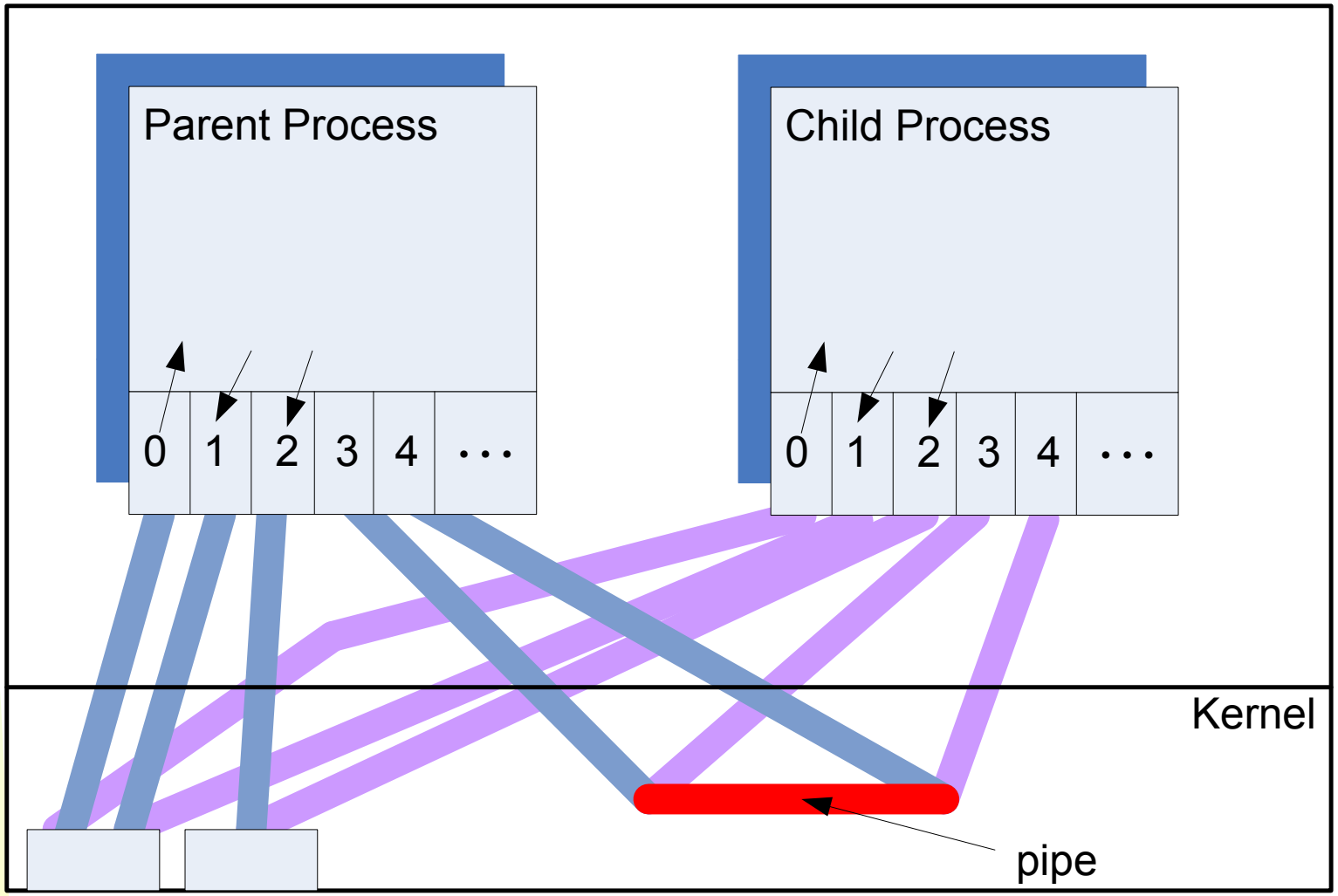
So, how do we make the parent talk to its child:

- The parent creates pipe
- Calls fork() to create child
- The parent closes the read end of the pipe and writes to the write end
- The child closes the write end of the pipe and reads from the read end
- Can we have the communication in the opposite way?

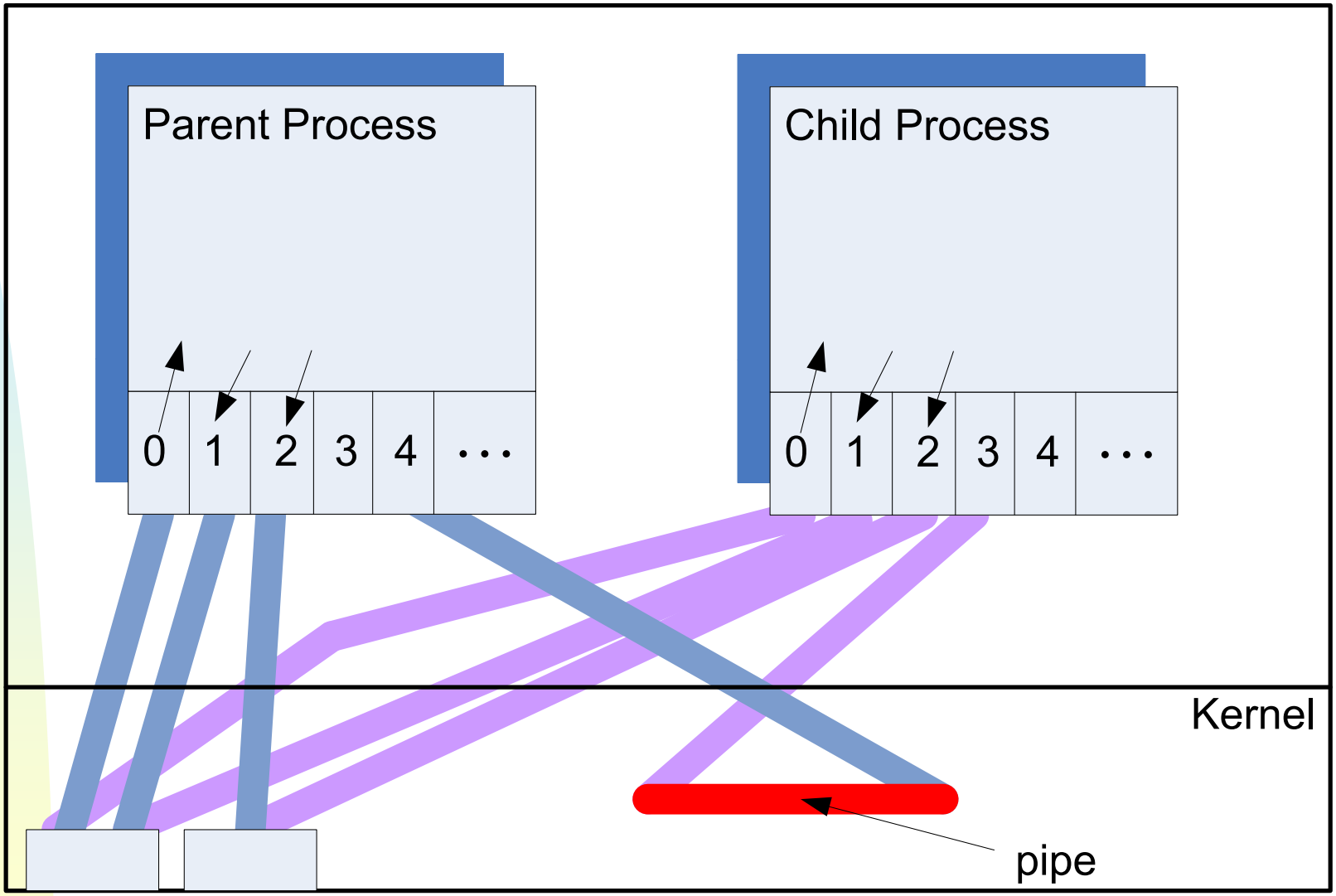
Unix Pipes

```
int fd[2], pid, ret;
ret = pipe(fd);
if (ret == -1) return PIPE_FAILED;
pid = fork();
if (pid == -1) return FORK_FAILED;
if (pid == 0) { /* child */
   close(fd[1]);
   while(…) {
       read(fd[0], …);
       …
   }
} else { /* parent */
   close(fd[0]);
   while(…) {
       write(fd[1], …);
       …
   }
}
```

After Spawning New Child Process

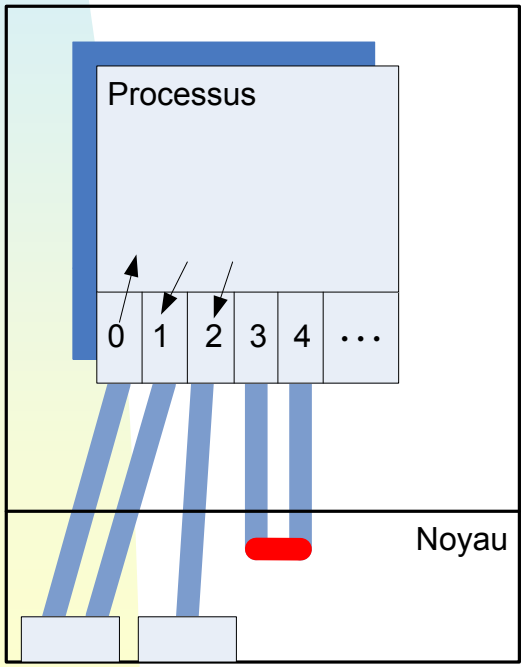


After Closing Unused Ends of Pipe

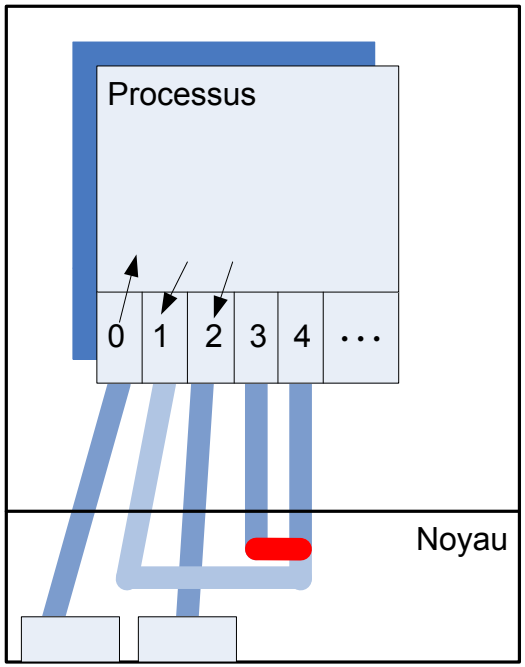


Pipes – attaching to 0 and 1

- System call: dup2(fdSrc,fdDst)
  - Creates a copy of fdSrc to fdDst
  - If fdDst is open, it is first closed
  - To attach a pipe to the standard output, dup2(fd[1],1)



(avant)



(après dup2)

- Note

  fd 4 still refers to the pipe

  It is possible to have multiple references to the ends of the pipe, including from other processes

Unix Pipes – Shell running who | sort

```
int fd[2], pid1, pid2, ret;
ret = pipe(fd);  // Create the pipe
if (ret == -1) return PIPE_FAILED;
pid1 = fork();  // for who
if (pid1 == -1) return FORK_FAILED;
if (pid1 == 0) { /* child */
  dup2(fd[1],1); // attach to stdout
  close(fd[1]);
  close(fd[0]);
  execlp("who","who",NULL);
}
pid2 = fork();  // for who
if (pid2 == -1) return FORK_FAILED;
if (pid2 == 0) { /* child */
  dup2(fd[0],0); // attach to stdout
  close(fd[0]);
  close(fd[1]);
  execlp("sort","sort",NULL);
}
if(wait(NULL) == pid1) printf("who finished")
else printf("sort finihed");
if(wait(NULL) == pid2) printf("sort finished")
else printf("who finihed");
```

Unix Pipes

Nice, nice, but we would like 2-way communication

- 2 pipes can be used

Gotcha's:

- Each pipe is implemented using a fixed-size system buffer
- If the buffer becomes full, the writer is blocked until the reader reads enough
- If the buffer is empty, the reader is blocked until the data is written
- What happens if both processes start by reading from pipes?
  - Deadlock, of course!
  - Similar with writing when the pipe becomes full
  - When working with several pipes, one has to be extra careful to avoid deadlock
  - A reader will block on the read end of the pipe as long as there is a reference to the write end – be careful about cleaning up unused references.

Unix Named Pipes

But we can still communicate only between related processes.

Solution: Named pipes

- Creating a named pipe (using `mkfifo()` system call) creates the corresponding file in the pipe file system
- Opening that file for reading returns a file descriptor to the read end of the pipe
- Opening that file for writing returns the write end of the pipe

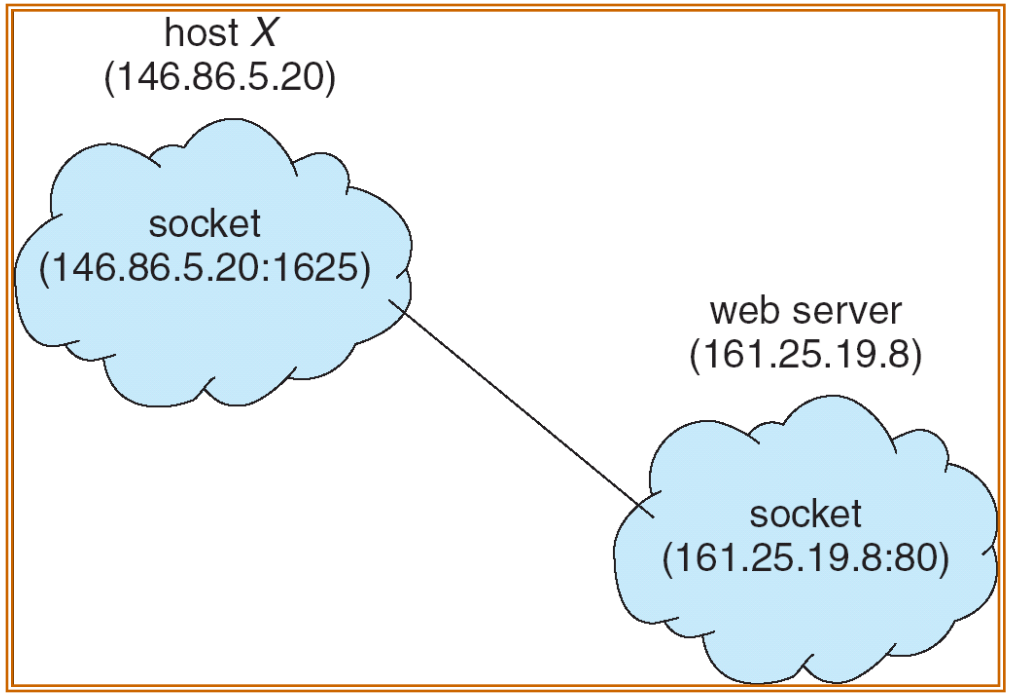
Unix Signals

- Similar to hardware interrupts without priorities
- Each signal is represented by a numeric value. Ex:
  - 02, SIGINT: to interrupt a process
  - 09, SIGKILL: to terminate a process
- Each signal is maintained as a single bit in the process table entry of the receiving process: the bit is set when the corresponding signal arrives (no waiting queues)
- A signal is processed as soon as the process runs in user mode
- A default action (eg: termination) is performed unless a signal handler function is provided for that signal (by using the *signal* system call)
- Want to know more?

  http://users.actcom.co.il/~choo/lupg/tutorials/signals/signals-programming.html

Sockets

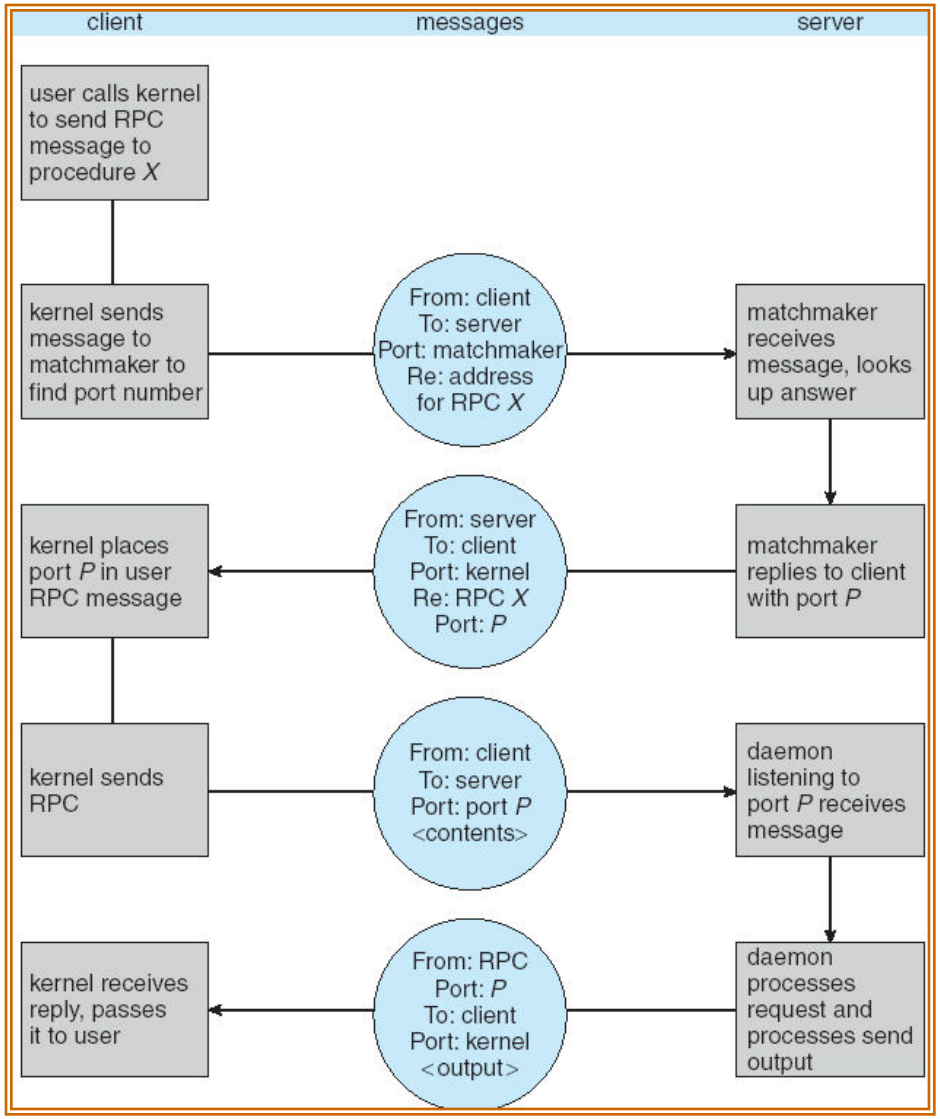
- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication link corresponds to a pair of sockets



Remote Procedure Calls

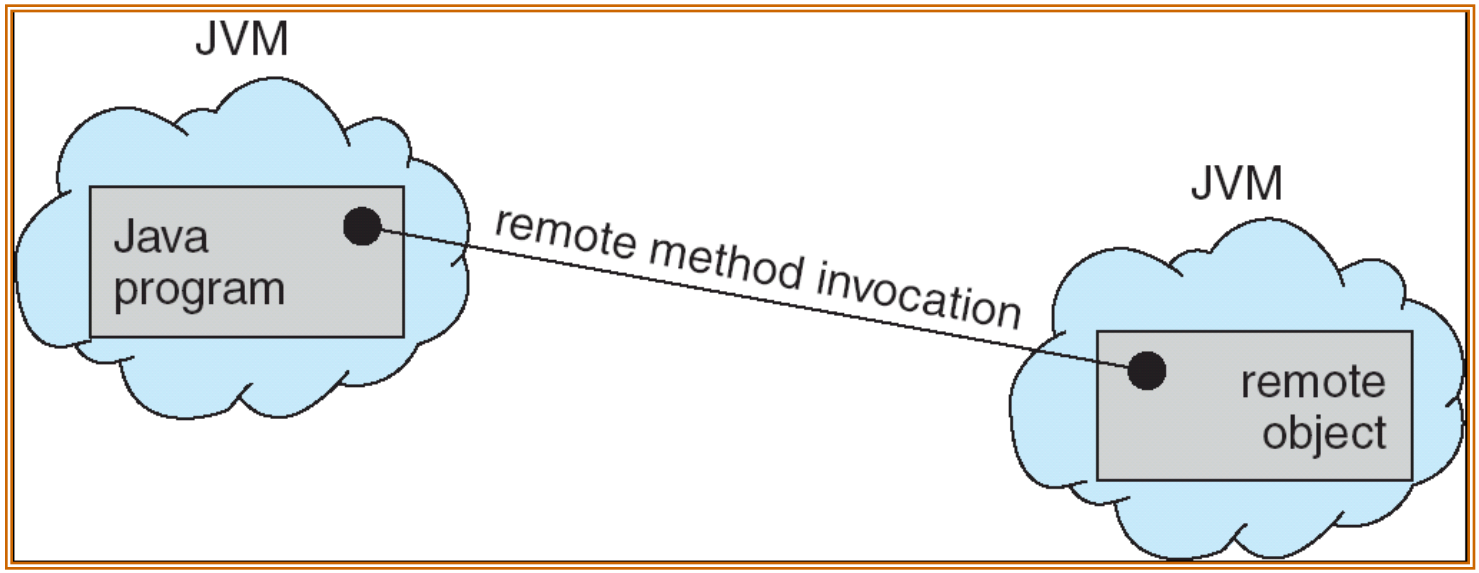
- **Wouldn't it be nice to be able to call procedures residing on other computers?**
- **Well, but the parameters and return values need to be somehow transferred between the computers.**
  - **The computers might have different data format**
- **Support for locating the server/procedure needed**
- Stubs **– client and server – side proxies implementing the needed communication**
- **The client-side stub locates the server and *marshalls* the parameters.**
- **The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server.**

Execution of RPC

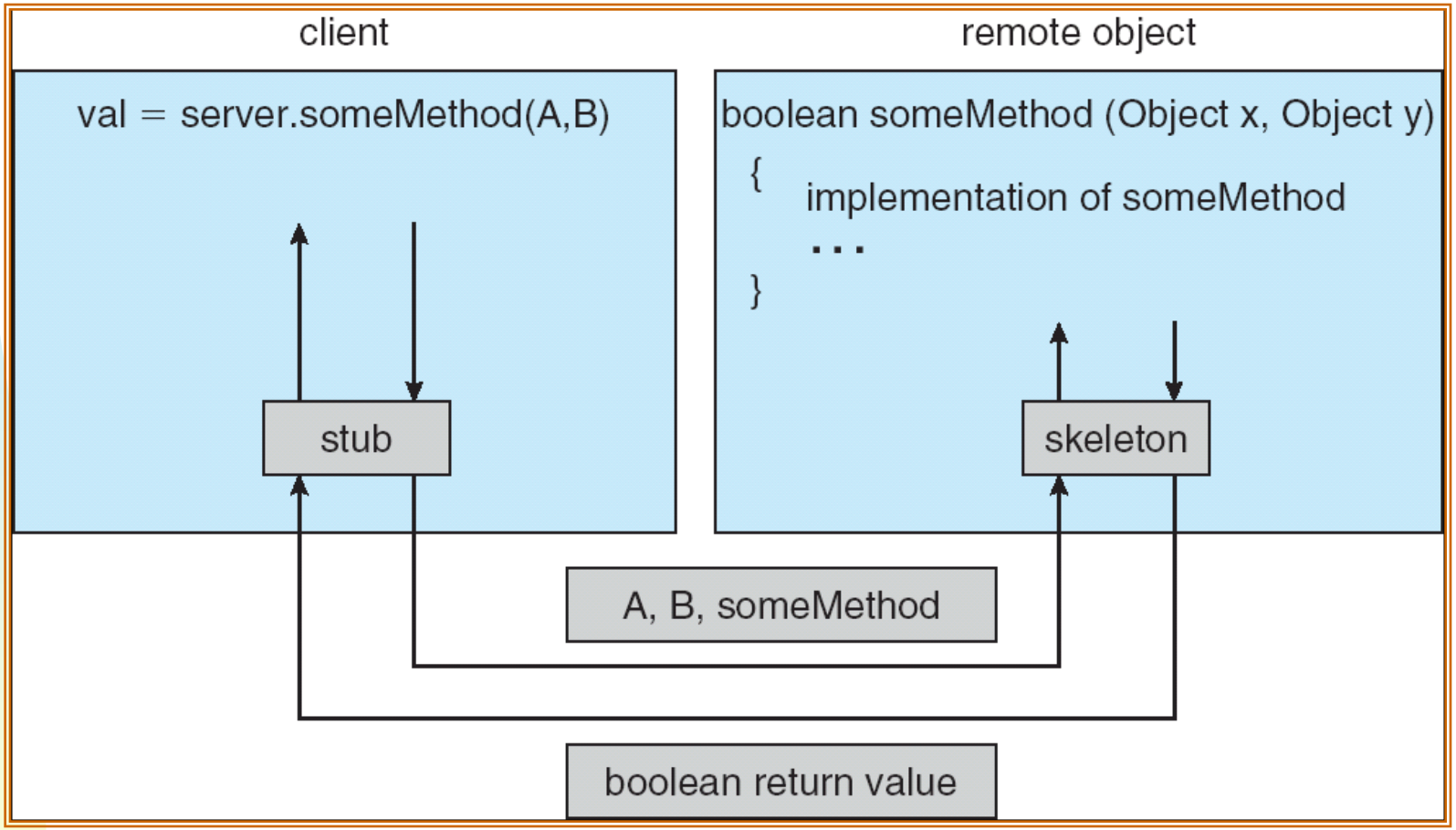


Remote Method Invocation

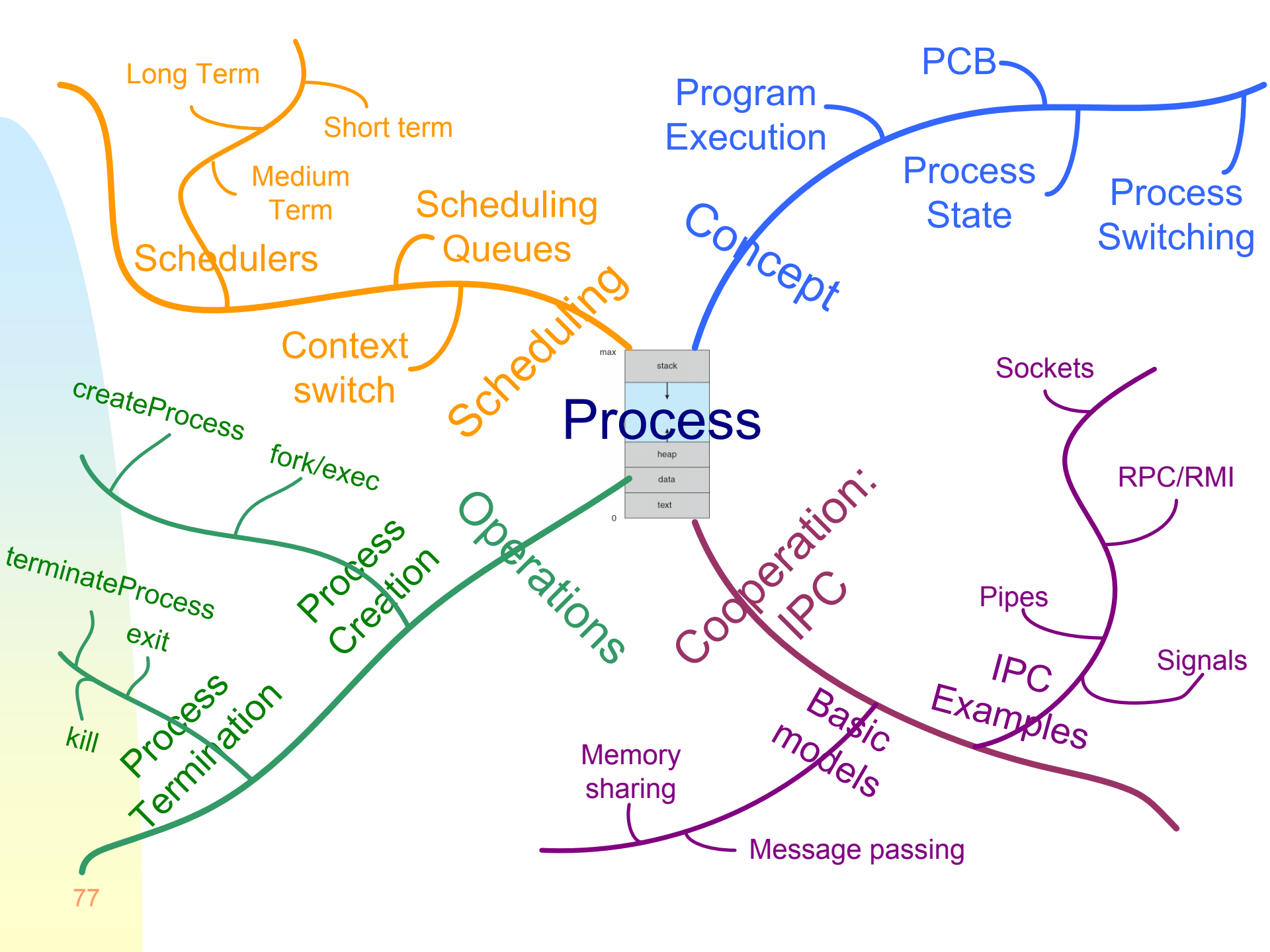
- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.



Marshalling Parameters



- **Local parameters: copies using « object serialization » technique**
- **Distant parameters: utilizes references.**


Process
Scheduling
Long Term
Short term
Medium Term
Schedulers
Scheduling Queues
Context switch
Concept
Program Execution
PCB
Process State
Process Switching
max
stack
heap
data
text
0
Operations
createProcess
fork/exec
Process Creation
terminateProcess
exit
kill
Process Termination
Cooperation: IPC
Sockets
RPC/RMI
Pipes
Signals
IPC Examples
Basic models
Memory sharing
Message passing