The process

CSI3131 - Module 2 - Processes

Reading: Chapter 3 (Silberchatz)

Goal:

Important concepts of Module 2 - Process

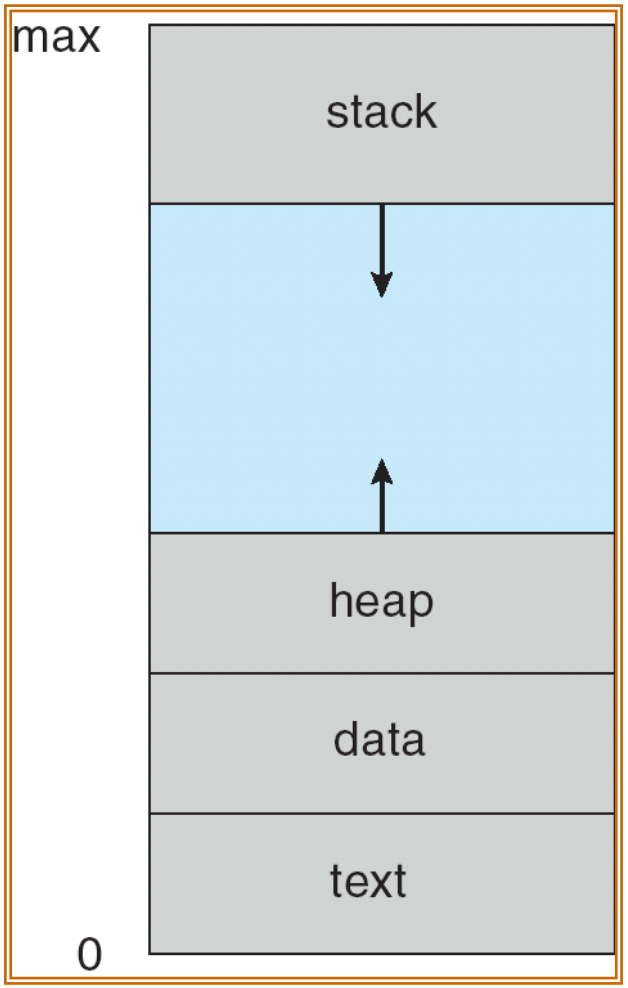
- **The process - to run a program**
  - **States and process state transitions**
  - **Process Control Block**
- **Process switching**
  - **PCB saving, reloading**
  - **Process queues and PCBs**
  - **Short, medium and long term schedulers**
- **Operations with processes**
  - Creation, termination, hierarchy
- **Communication between processes**

Process and terminology (also called job, task, user program)

- Process concept: a program in execution
  - Has memory resources, peripherals, etc.
- Process scheduling
- Process operations
- Example of process creation and termination
- Cooperating processes (communication between processes)

Process runs a program

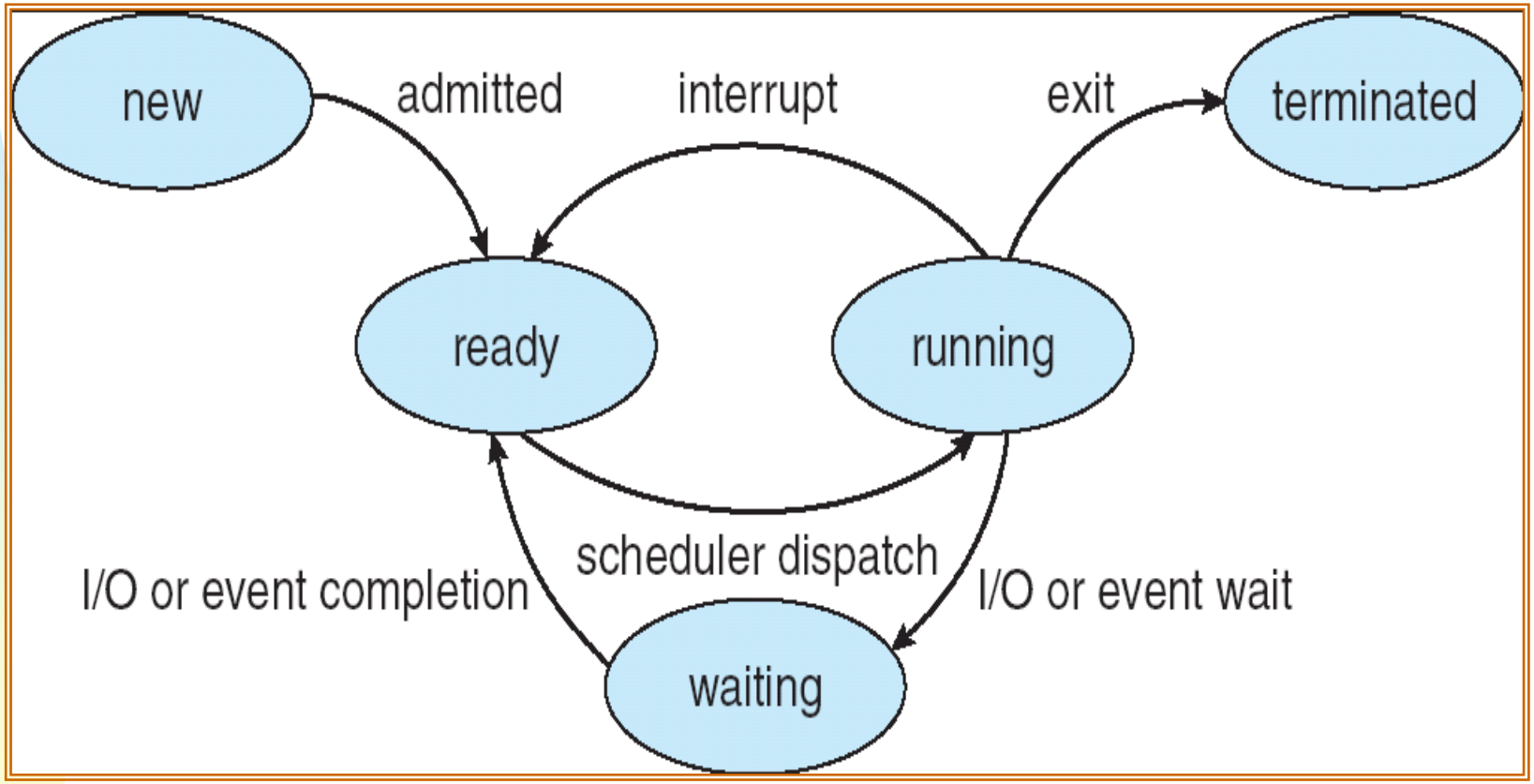
- Text - code to run
- Data
- Heap
- Stack



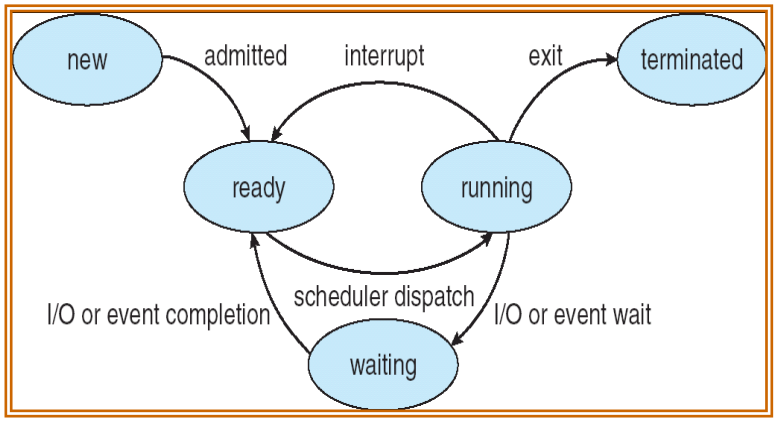
process state *IMPORTANT*

- As a process executes, it changes state
  - new: the process has just been created
  - Exécution - running: the process is being executed by the CPU
  - waiting: the process is waiting for an event (e.g. the end of an I / O operation)
  - ready: the process is waiting to be executed by the CPU
  - terminated: end of execution

State transition diagram of a process

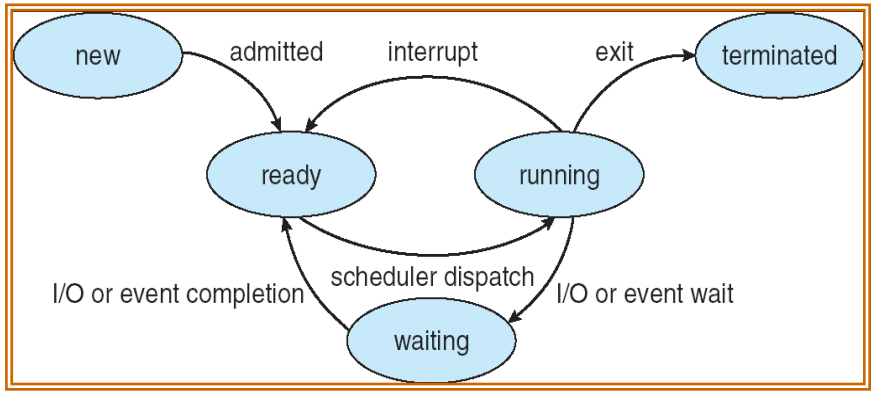


Transitions between processes



- **Ready → Running**
  - **When the CPU scheduler chooses a process for execution**
- **Running → Ready**
  - **Result of an interruption caused by a process independent event**
    - **This interrupt must be treated, so the current process loses the CPU**
      - Important case: the process has exhausted its time interval (timer)

Transitions between processes



- **Running → Waiting**
  - **When a process requests a service from the OS that the OS cannot immediately offer**
    (interruption caused by the process itself)
    - **access to a resource not yet available**
    - **initiate I/O: must wait for the result**
    - **needs response from another process**
- **Waiting → Ready**
  - **when the expected event occurs**

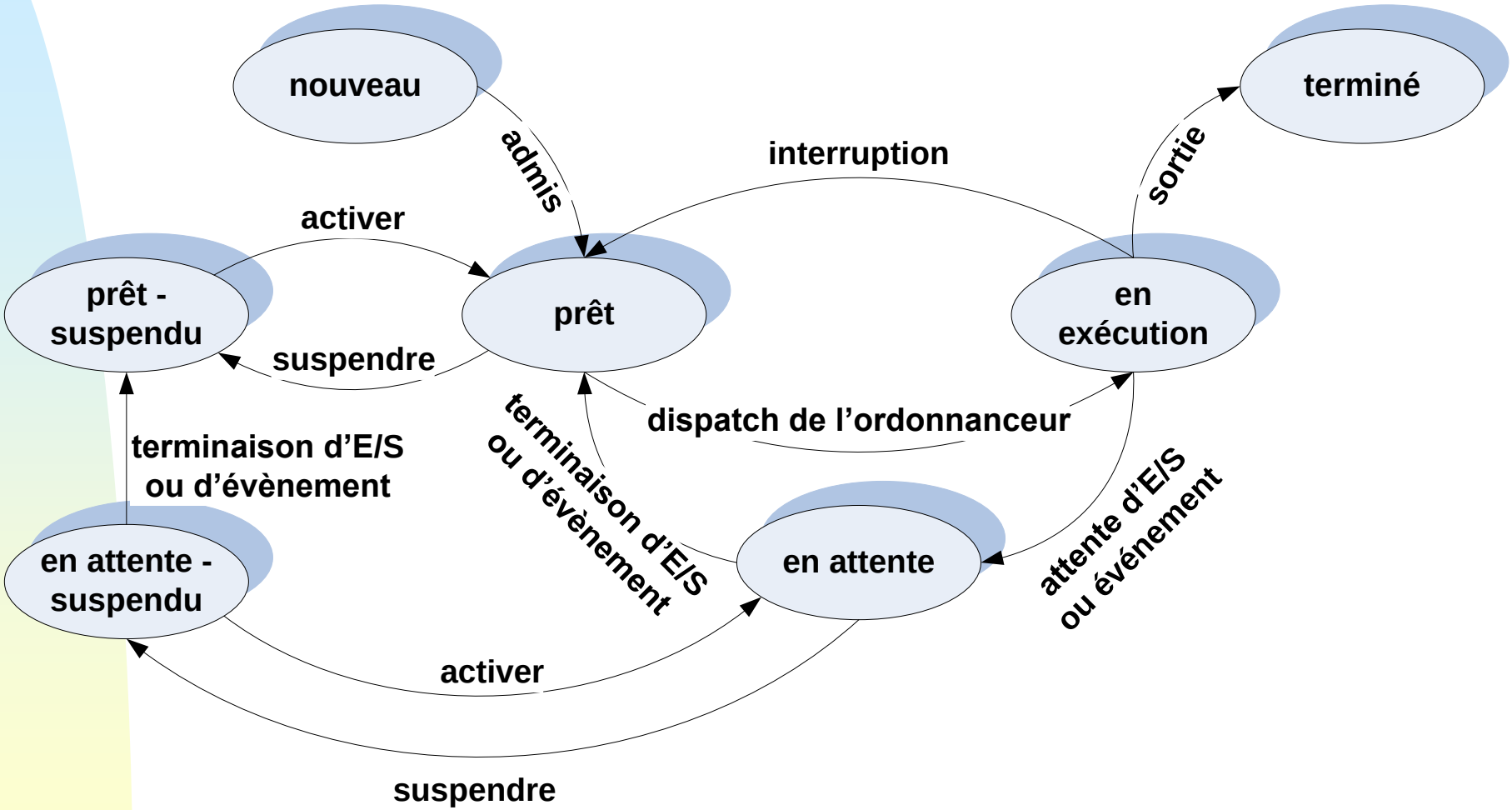
The need to replace (swapping)

- **Until now, all processes were (at least partially) in memory (RAM)**
- **Even with virtual memory, the OS cannot keep too many processes in memory without deteriorating performance**
- **Sometimes the OS must suspend some processes, ie: transfer them to disk. So 2 other states:**
- **Waiting Suspended: blocked processes transferred to disk**
- **Ready Suspended: ready processes transferred to disk**

New transitions

- Waiting -> Waiting Suspended
  - Preferred choice by the OS to free memory
- Waiting Suspended -> Ready Suspended
  - When the expected event occurs (status info is available to the OS)
- Ready Suspended -> Ready
  - when there is no more process ready (in memory)
- Ready -> Ready Suspended (rare)
  - We must free some memory but there are no more blocked processes in memory

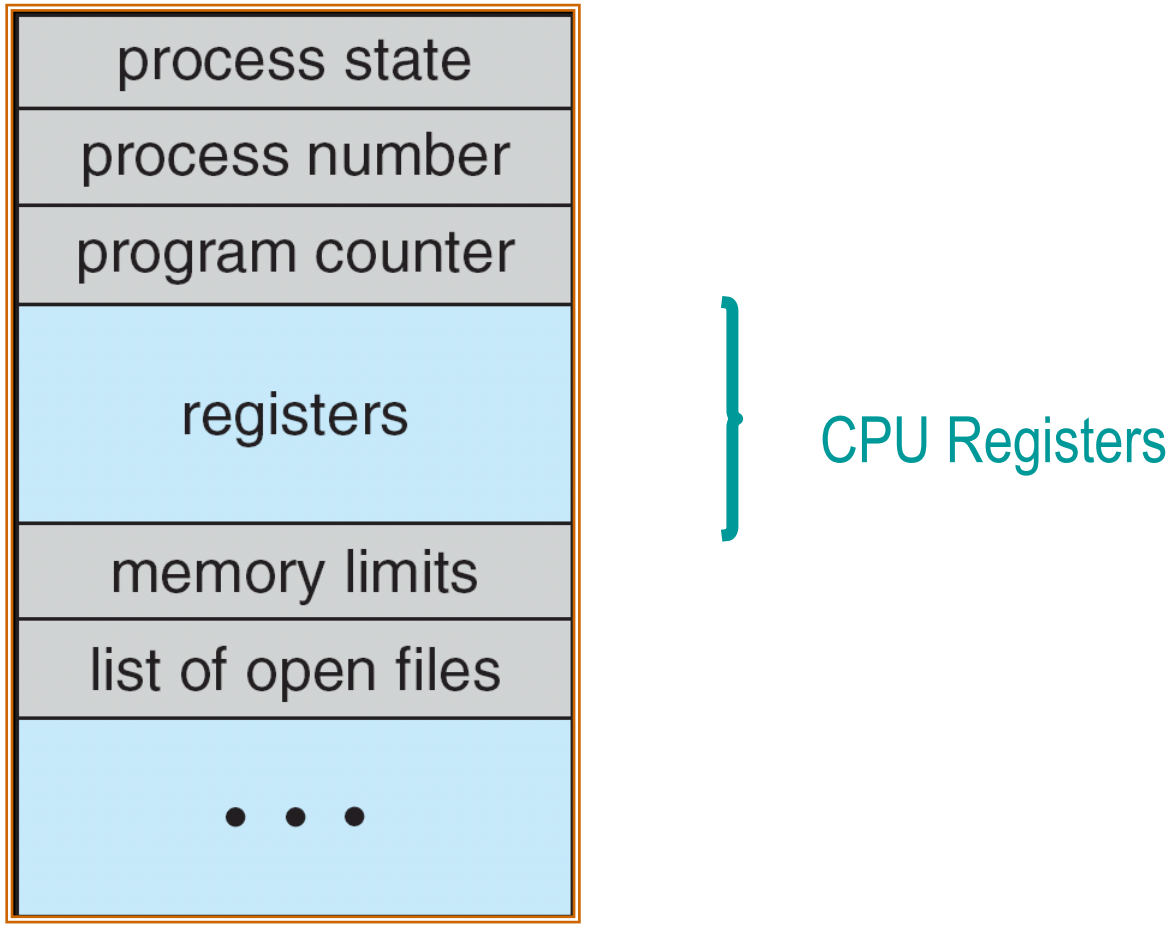
A 7-state process model



Saving process information

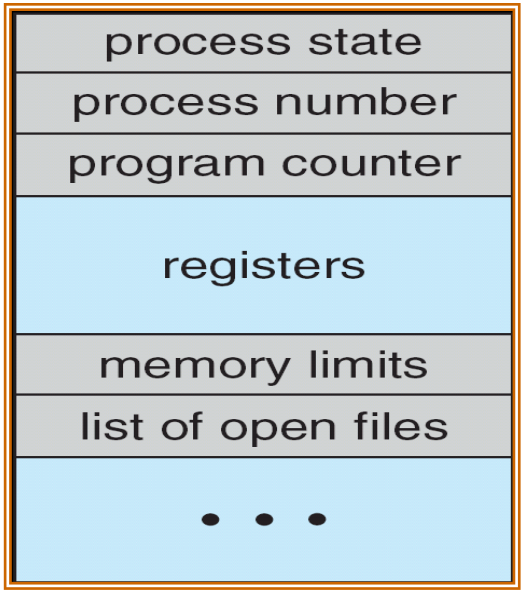
- In multiprogramming, a process is running on the CPU intermittently
- Whenever a process takes over the CPU (transition ready → execution) it must resume it in the same situation where it left it (same content of CPU registers, etc.)
- So when a process exits the execution state, it is necessary to save its essential information, which will have to be recovered when it returns to this state.

PCB = Process Control Block:
Represents the current status of a process, to resume it later



Process Control Block (PCB)
IMPORTANT

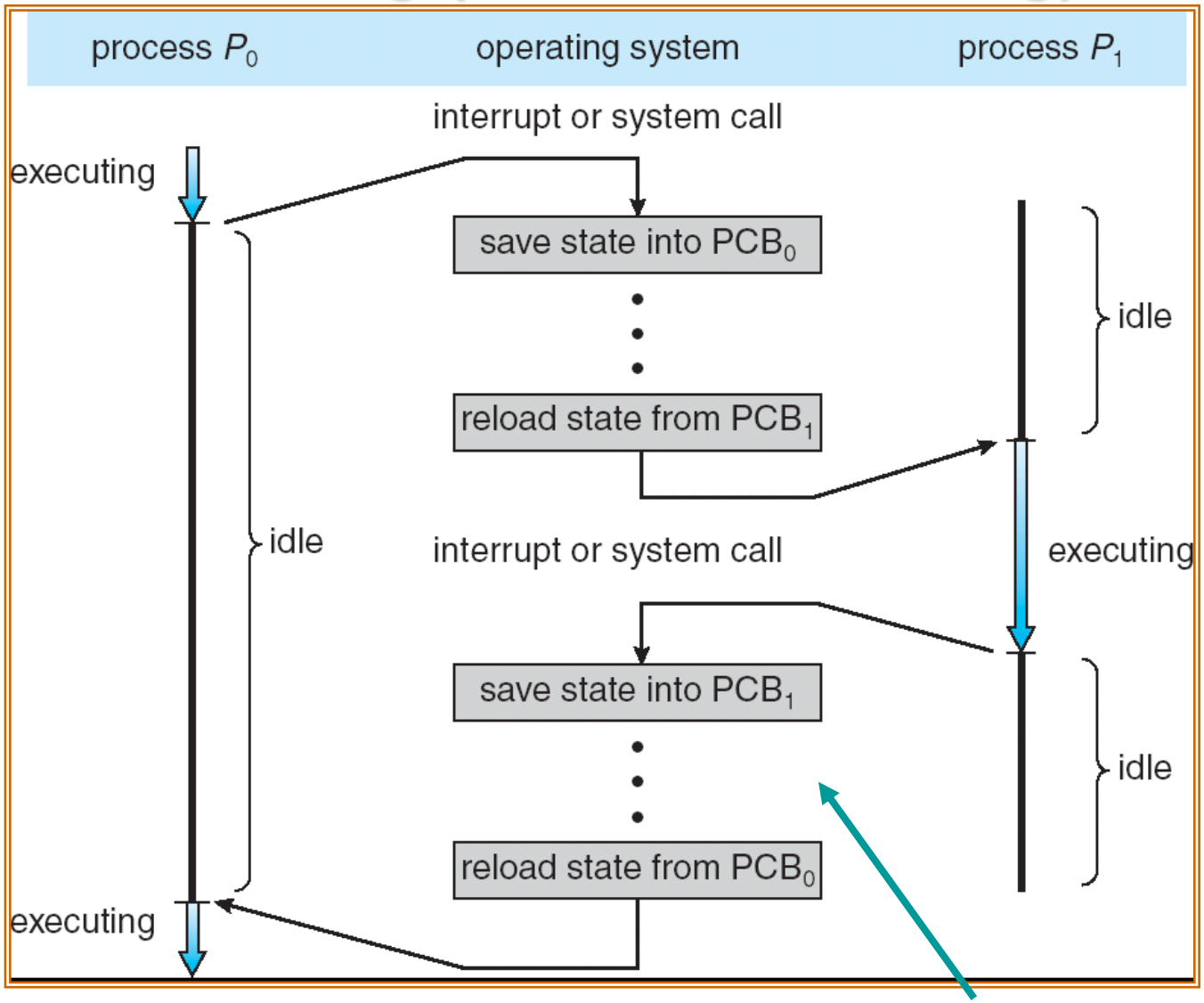
- pointer: the PCBs are arranged in linked lists (to see)
- process state: ready, running, waiting ...
- program counter: the process must resume at the next instruction
- other CPU registers
- memory limits
- Memory management
- Opened files
- etc., see. textbook



Process switching
Also called context switching

- When the CPU switches from executing a process 0 when running a proc 1, it is necessary
  - update the PCB of 0
  - save the PCB of 0
  - take back the PCB from 1, which had been saved before
  - reset CPU registers, instruction counter etc. in the same situation which is described in the PCB of 1

Processor switching (context switching)



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

The PCB is not the only information to save ... (the textbook is not clear here)

- It is also necessary to save the state of the program data
- This is normally done by keeping the *'program image* in primary or secondary memory
- The PCB will point to this image

Process and terminology (also called job, task, user program)

- Process concept: a program in execution
  - Has memory resources, peripherals, etc.
- Process scheduling
- Process operations
- Example of process creation and termination
- Cooperating processes (communication between processes)

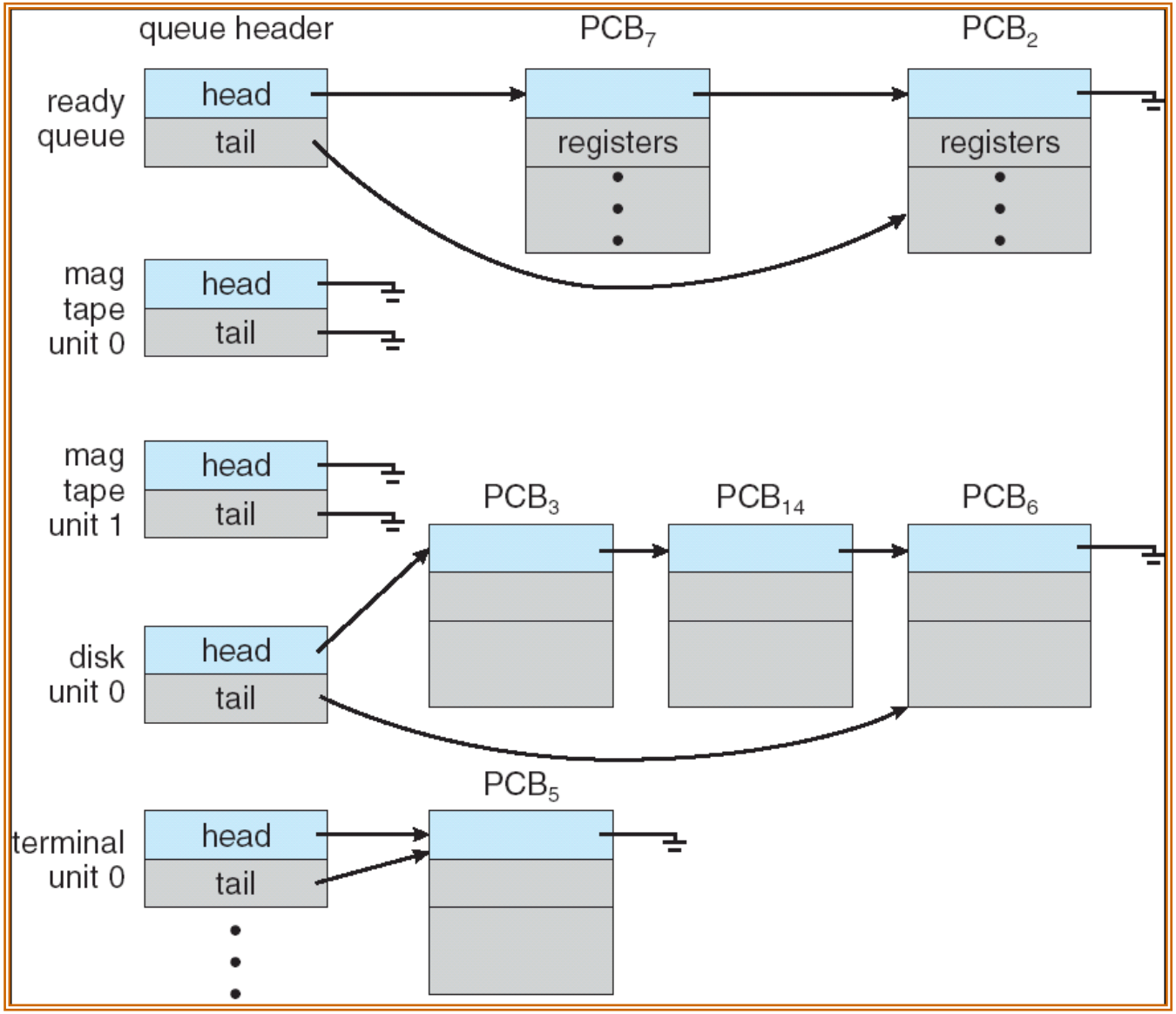
Scheduling of processes

- What's this?
  - Selecting the next process to run
- Why?
  - Multiprogramming - to achieve good system use
  - Time sharing - To allow a good response time
- How? 'Or' What?
  - In detail in Module 4 (Chapter 5 of the text)
  - Now introduce the fundamentals of scheduling
  - Already introduced first: process switching

Waiting Queues *IMPORTANT*

- Computer resources are often limited compared to the processes that require them.
- Each resource has its own queue of pending processes
- By changing state, processes move from one queue to another
  - Ready Queue: processes in ready state = ready
  - Queues associated with each I / O unit
  - etc.

Ready Queue And Various I/O Device Queues



A more synthetic way of describing the same situation

ready → 7 → 2

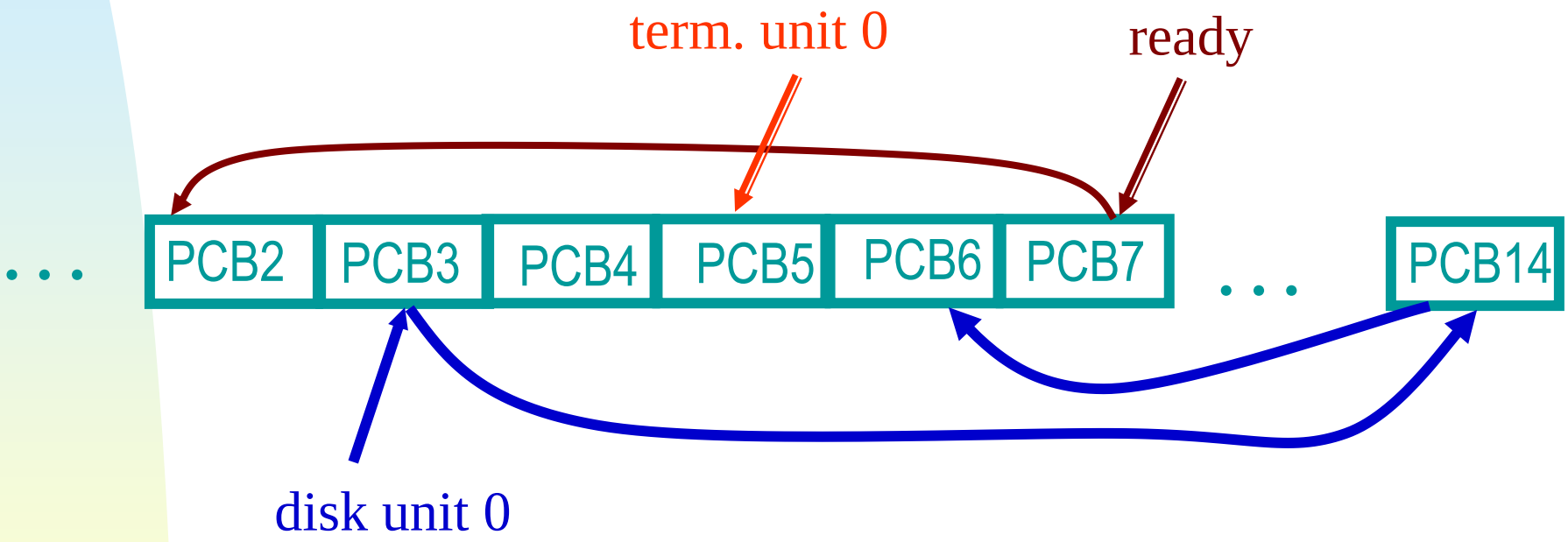
magtape0 →

magtape1 →

disk0 → 3 → 14 → 6

term0 → 5

PCBs are not moved in memory to be put in the different queues: it's the pointers that change.



Schedulers

- Programs that manage the use of computer resources
- Three types of schedulers:
  - Short term = process scheduler: select which process should execute the transition ready → execution
  - Long term = job scheduler: select which processes can execute the transition new → ready (event *admitted*) (from spool jobs to ready queue) (for batch processing system, *batch*)
  - medium term: we will see (timesharing systems)

Schedulers

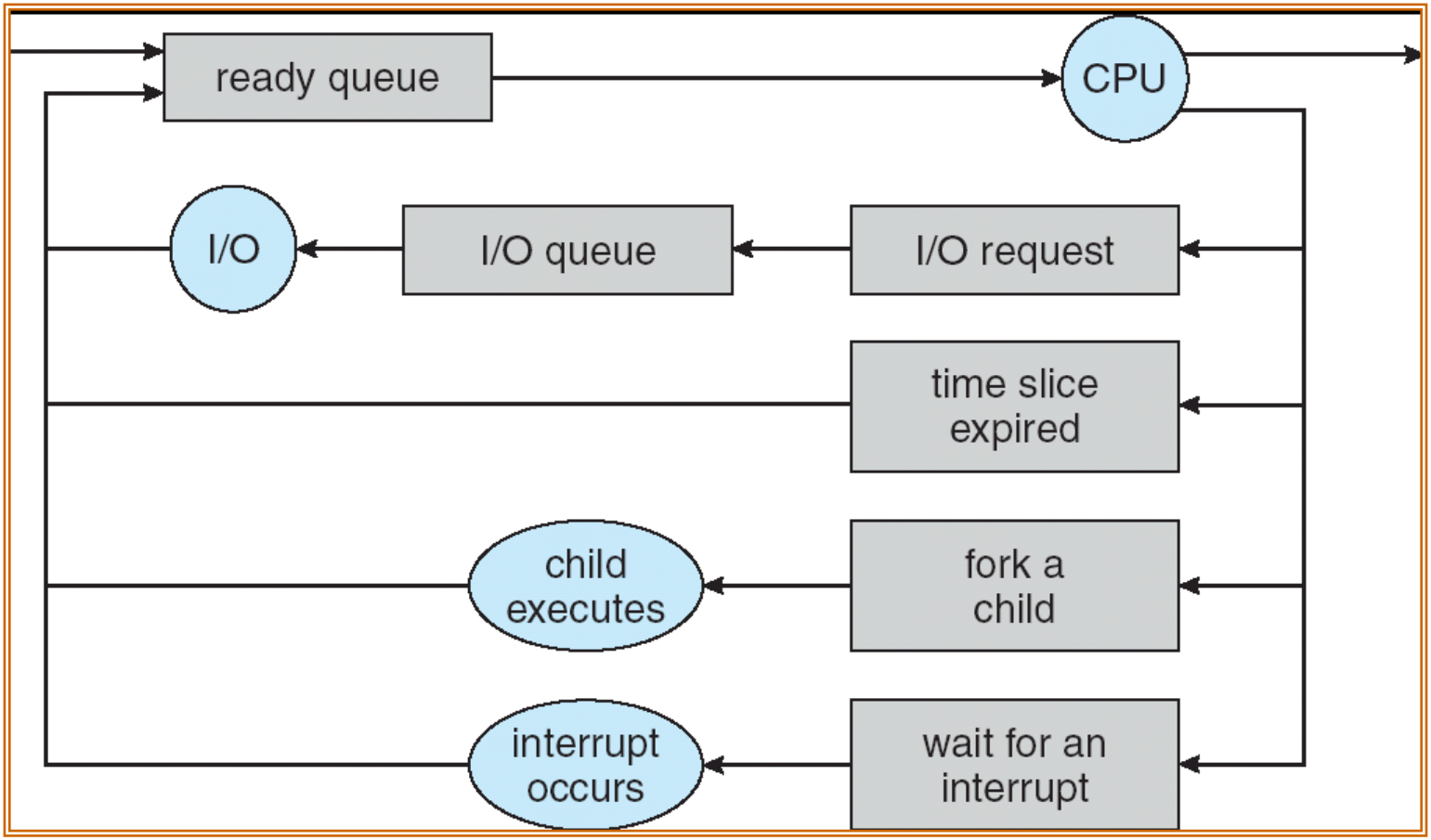
- Long-term scheduler **(or job scheduler)**
  - **selects which new processes should be brought into the memory; new → ready (and into the ready queue from a job spool queue) (used in batch systems)**
- Short-term scheduler **(or CPU scheduler)**
  - **selects which ready process should be executed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

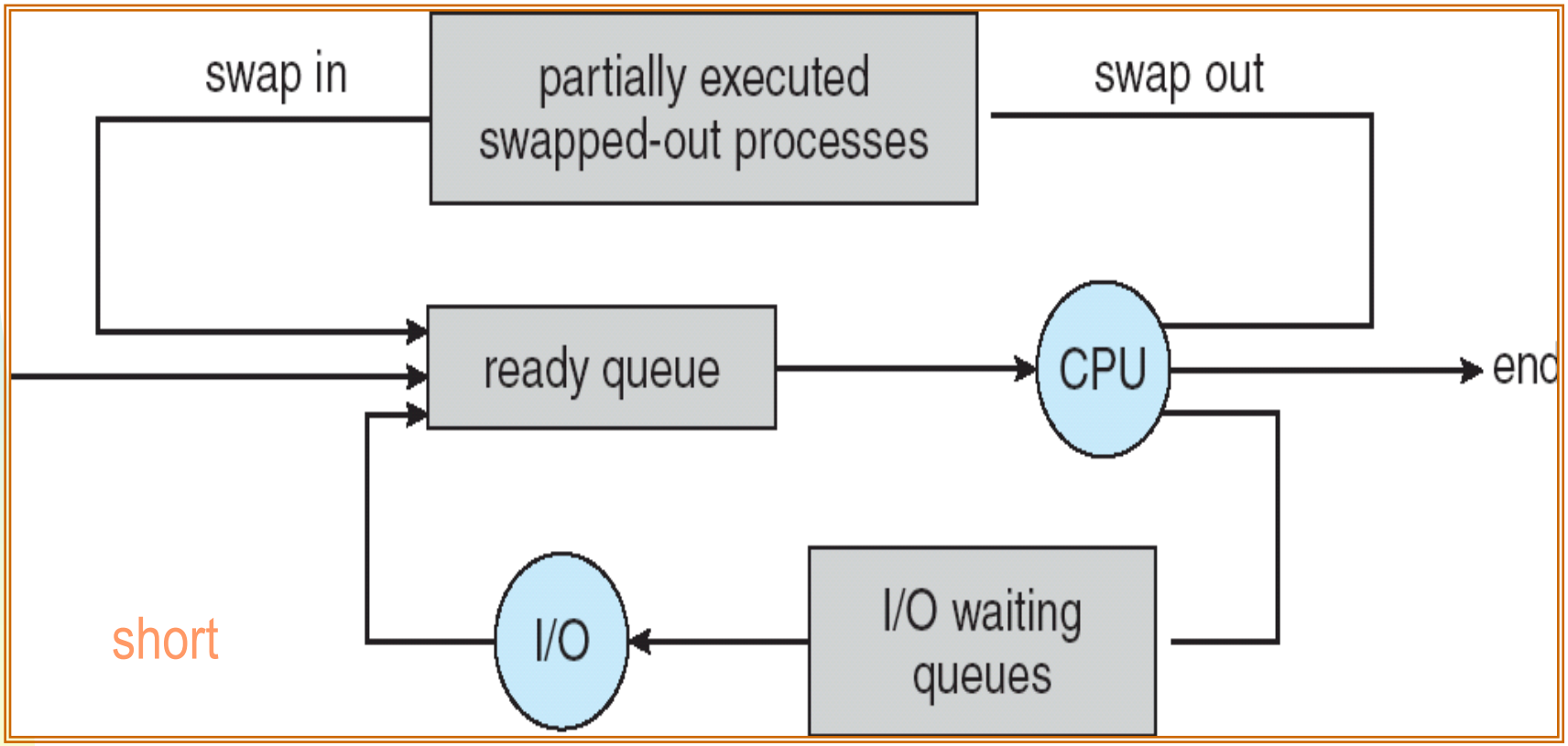
Process Scheduling (Short Term)



Medium-term scheduler

- **Lack of resources can sometimes force the OS to *suspend* process**
  - **they will be more in competition with others for resources**
  - **they will be resumed later when the resources become available**
- **These processes are removed from main memory and placed in secondary memory, to be resumed later.**
  - **`swap out`,` swap in`, back and forth**

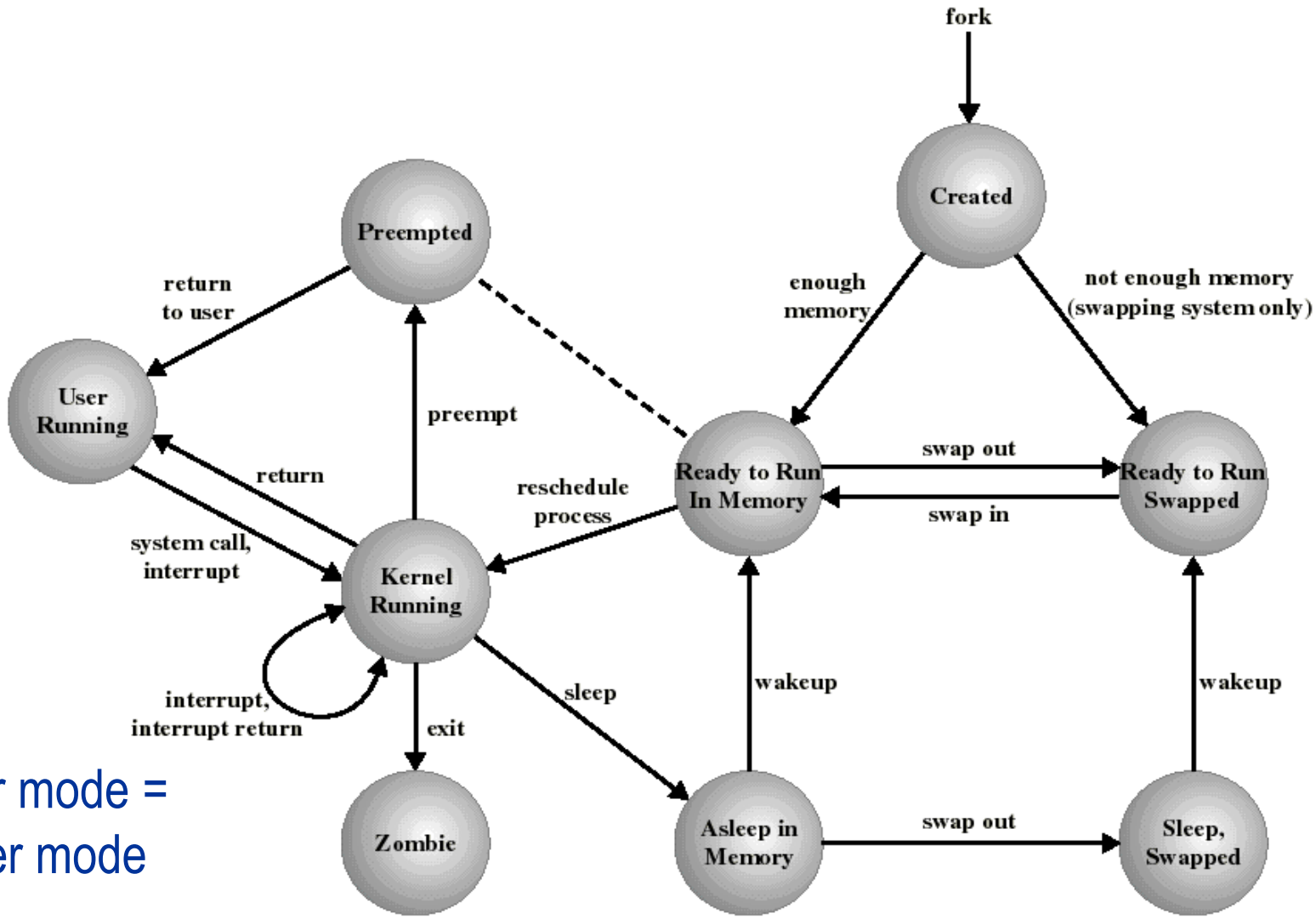
short and medium term Schedulers



edium

Process States in UNIX SVR4 (Stallings)

An example of a state transition diagram for a real OS



Kernel, user mode = monitor, user mode

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 smartness)**

Process and terminology (also called job, task, user program)

- Process concept: a program in execution
  - Has memory resources, peripherals, etc.
- Process scheduling
- Process operations
- Example of process creation and termination
- Cooperating processes (communication between processes)

Process Creation

So, where do all processes come from?

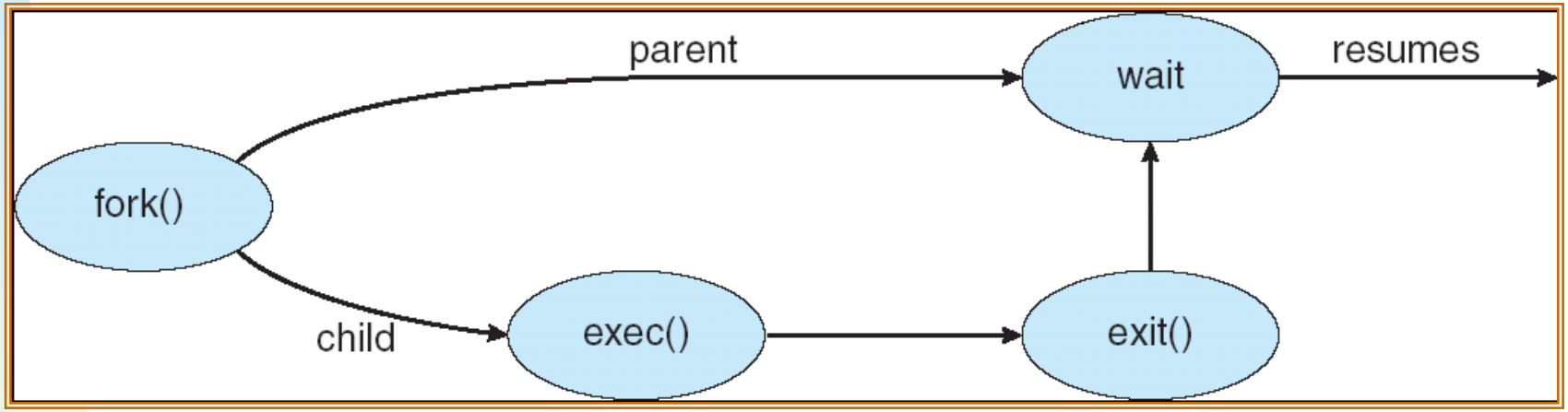
- **Parent process create children processes, which, in turn create other processes, forming a tree of processes**
- **Usually, several properties can be specified at child creation time:**
  - **How do the parent and child share resources?**
    - Share all resources
    - Share subset of parent's resources
    - 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);
    exit(0);
}
```

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

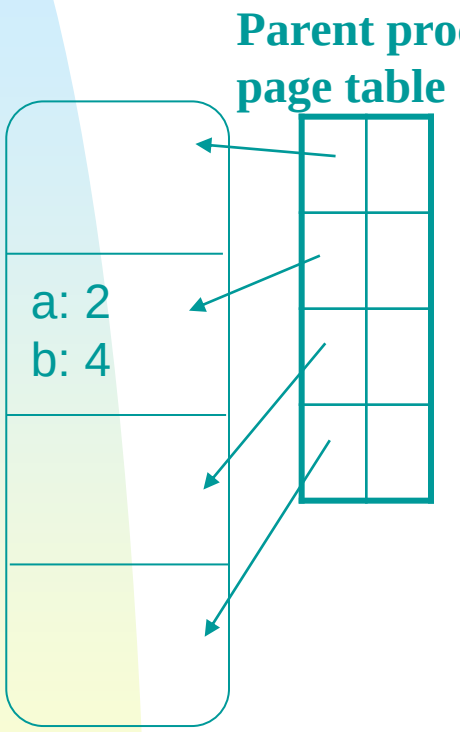
```
int pid, a = 2, b=4;
pid = fork();    /* fork another process
*/
if (pid < 0)  exit(-1); /* fork failed */
else if (pid == 0) { /* child process */
    a = 3; printf("%d\n", a+b);
} else {
    b = 1;
    printf("%d\n", a+b);
    wait();
}
```

What would be the output printed?

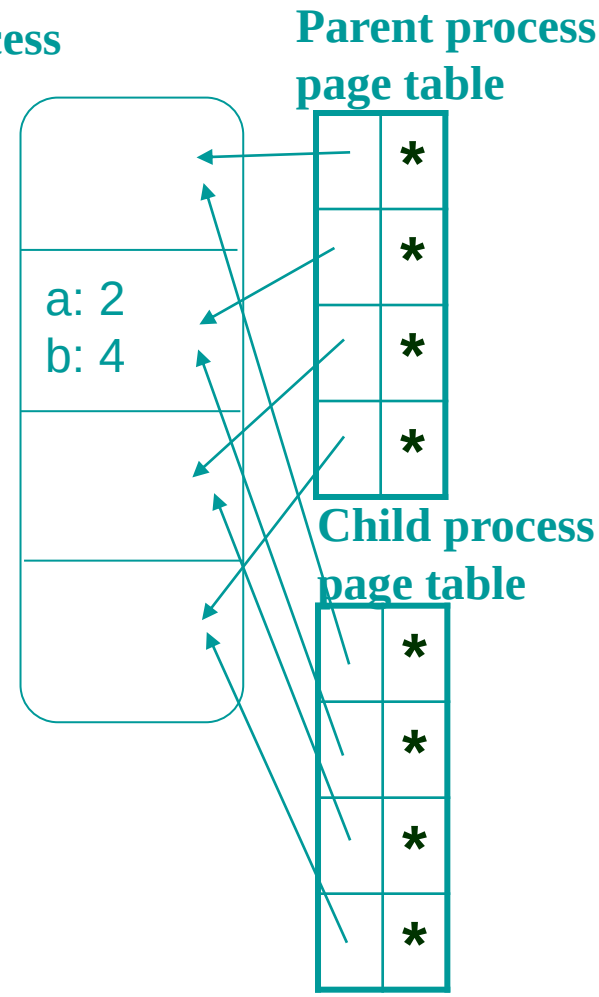
7	or	3
3		7

Understanding fork()

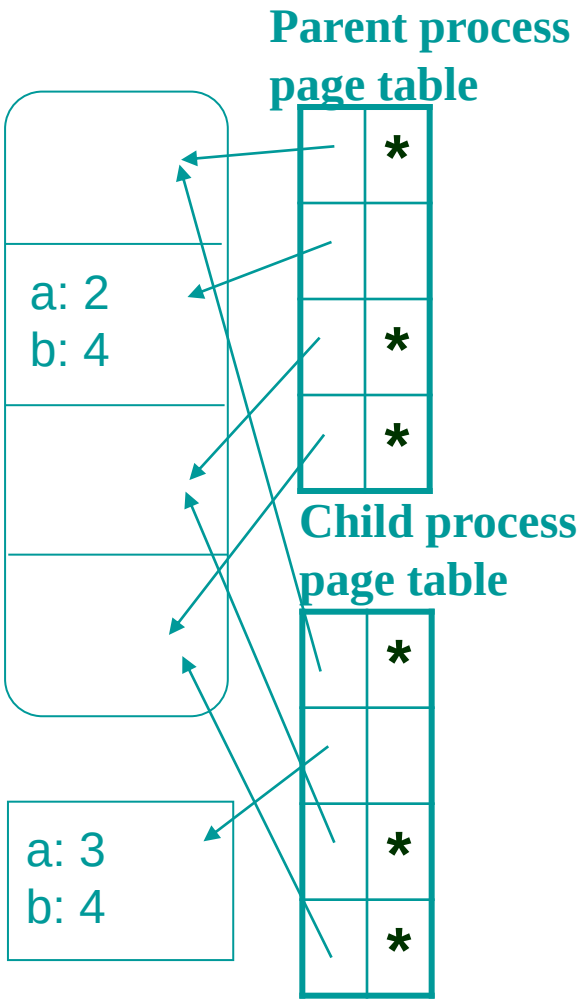
Before fork



After fork



After child executes a=3



Process Creation (Cont.)

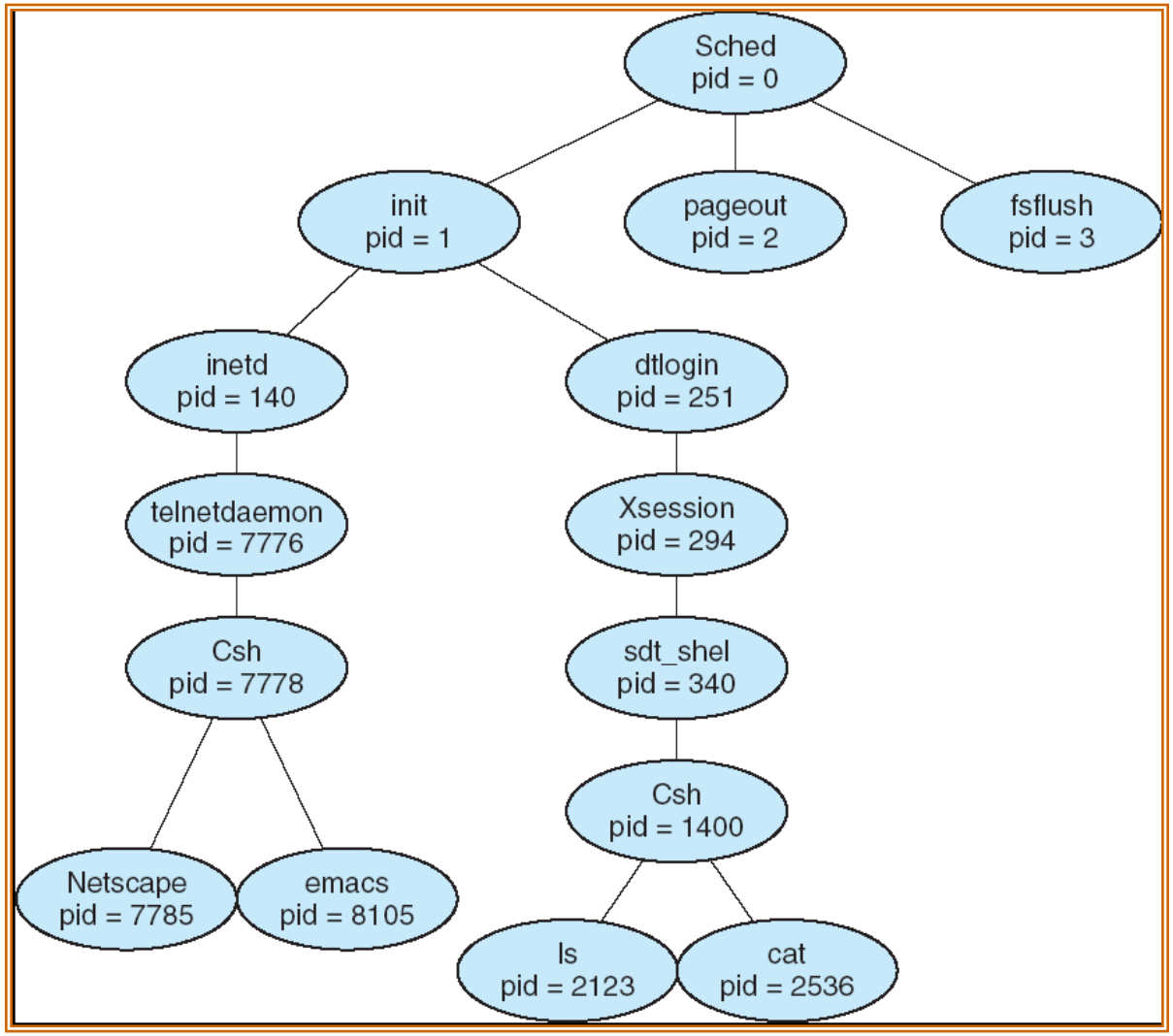
Windows:

- 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 of a process tree (Solaris)



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 waiting state?**

PCB

- **Does PCB contain program's active (running) time?**
- **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()**
- **Test if you really understand them**
  - **So, how many processes are created in this code fragment?**

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

8 processes exist the end (including the very first parent process)

Process termination

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

Process and terminology (also called job, task, user program)

- Process concept: a program in execution
  - Has memory resources, peripherals, etc.
- Process scheduling
- Process operations
- Example of process creation and termination
- Cooperating processes (communication between processes)

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

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**

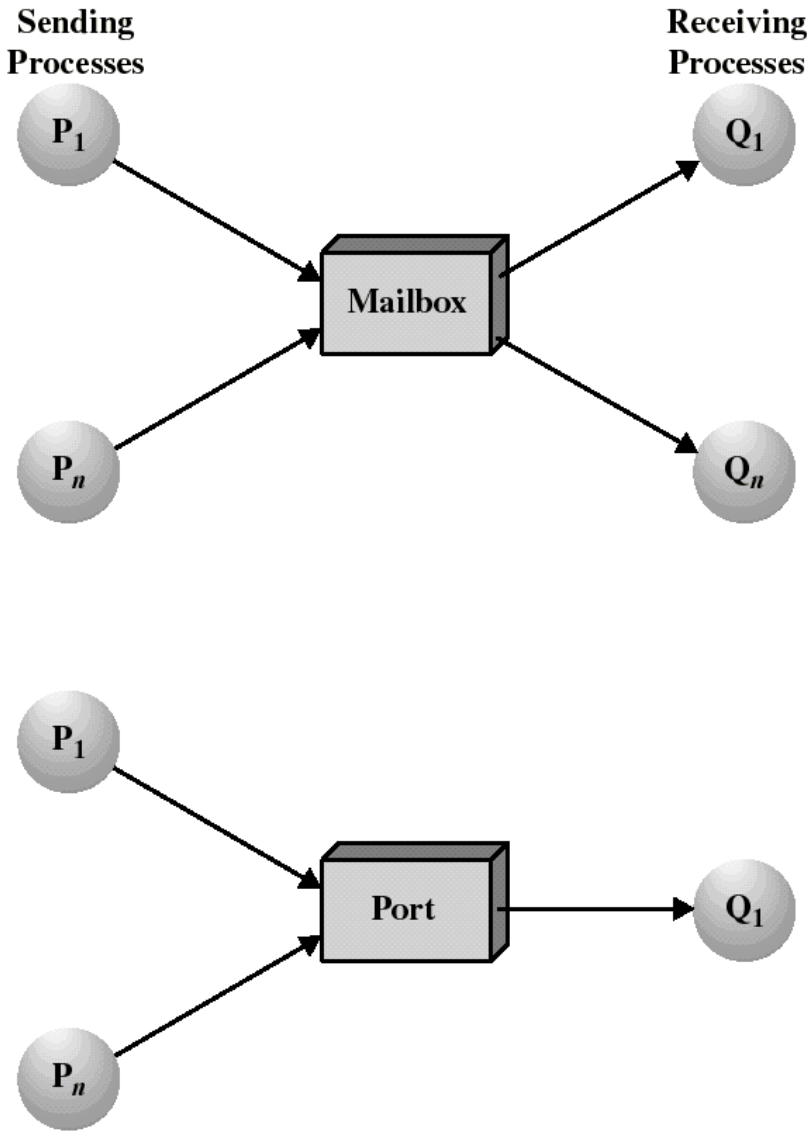
Indirect Communication

- **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.

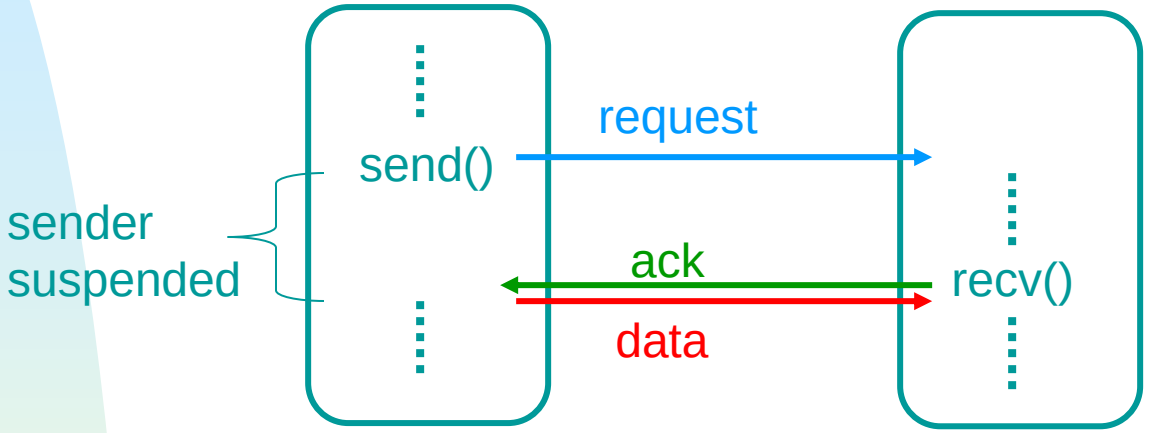
Blocking Message Passing

Also called synchronous **message passing**

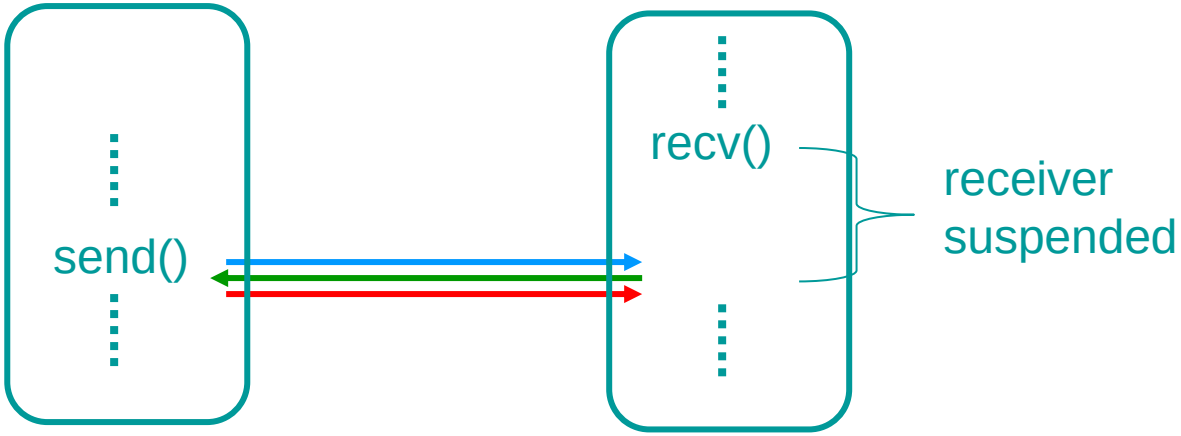
- **sender waits until the receiver receives the message**
- **receiver waits until the sender sends the message**
- advantages:
  - **inherently synchronizes the sender with the receiver**
  - **single copying sufficient**
- disadvantages:
  - **possible deadlock problem**

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

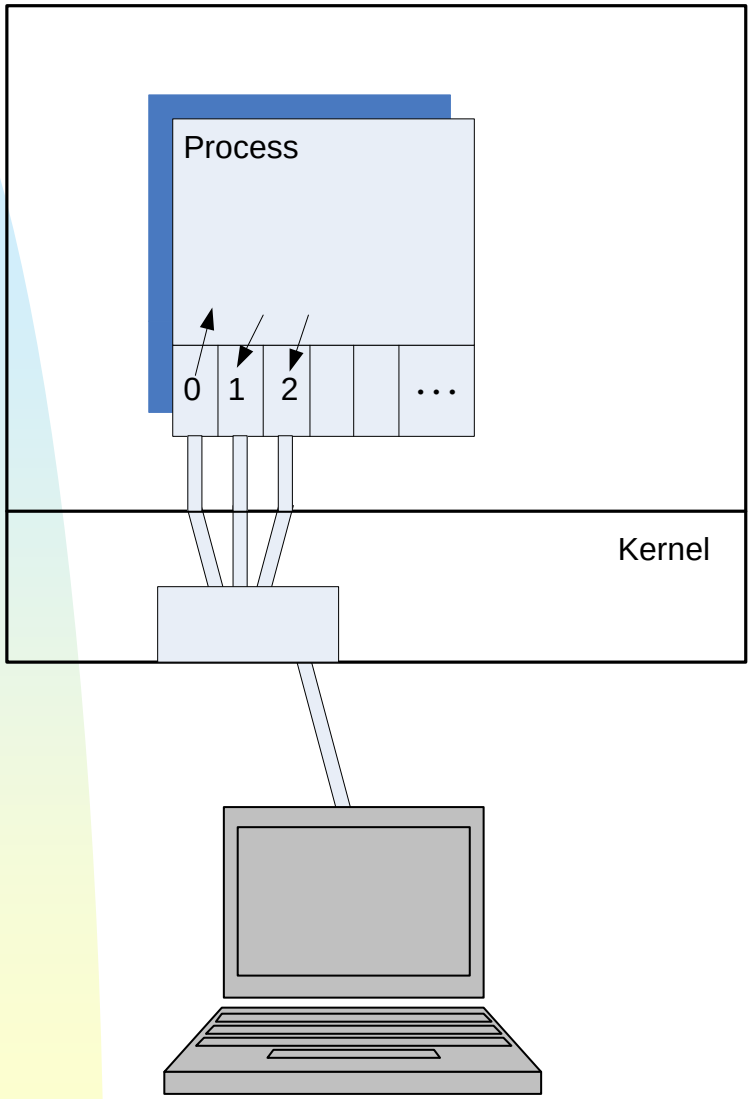
- With blocking direct communication, no buffering is needed – the message stays in the sender's buffer until it is copied into the receiver's buffer
- With non-blocking communication, the sender might reuse the buffer,
  - therefore the message is copied to a system buffer,
  - and from there to the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

Examples of IPC Mechanisms

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

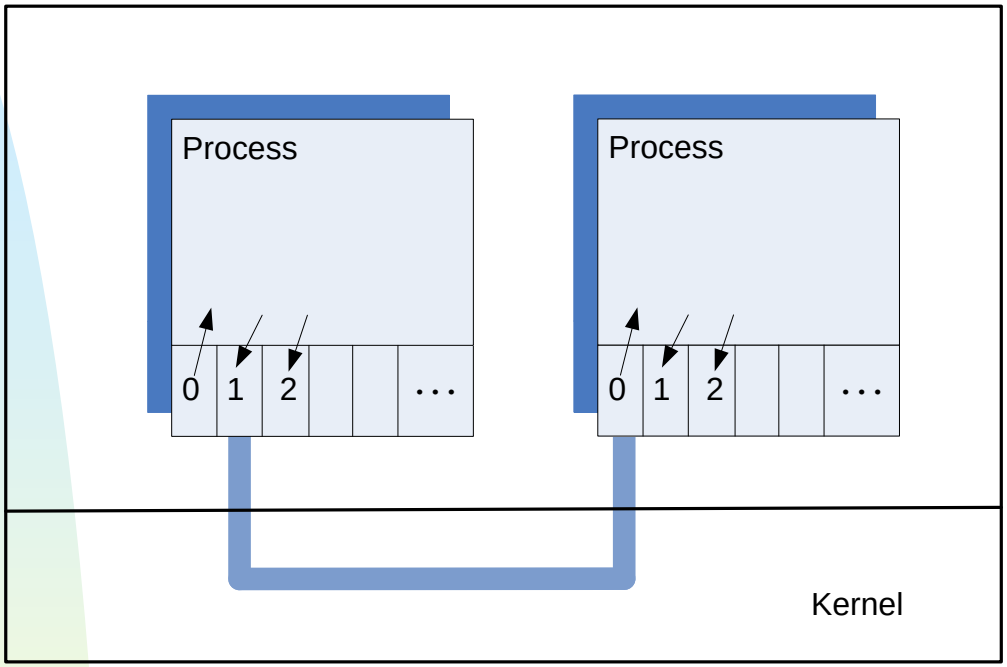
- Higher-level concepts
  - Implemented either using shared memory (between processes on the same computer) or message passing (between different computers)

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

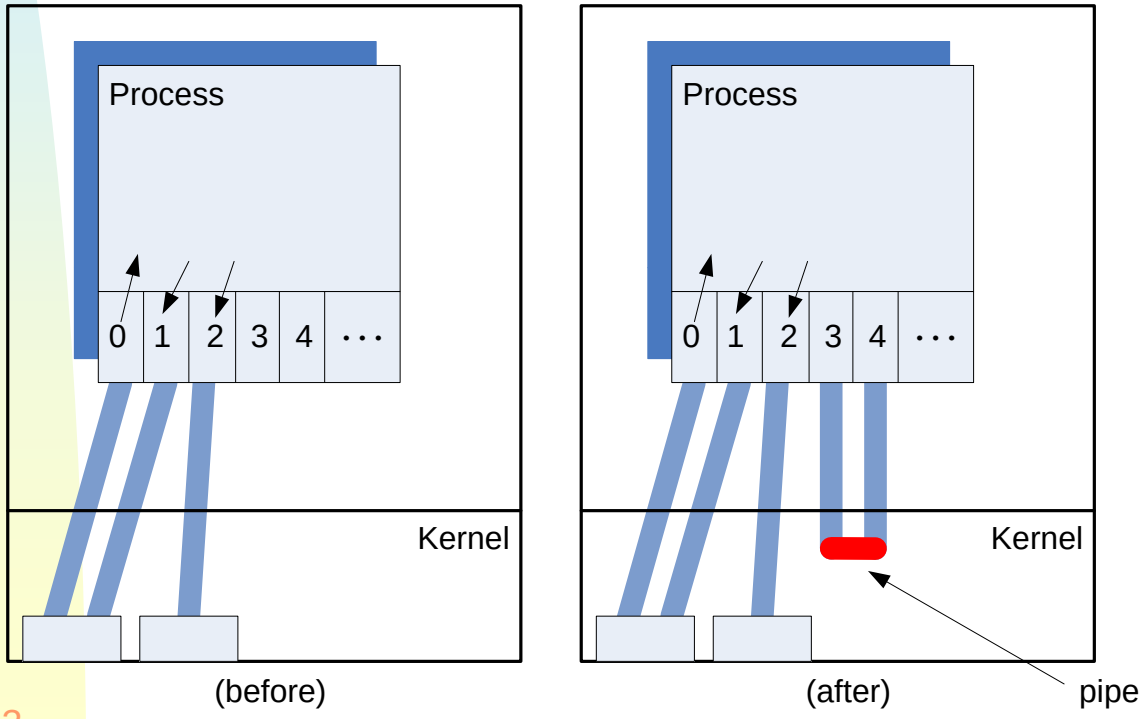


- **What is it?**
  - **A unidirectional channel**
  - **One write end**
  - **One read end**

- **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[2])**
  - **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(pipes[0], dataBuf, count) would read count bytes from the pipe into the dataBuf character array
- fd[1] is a file descriptor for the write end of the pipe
  - write(pipes[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.**

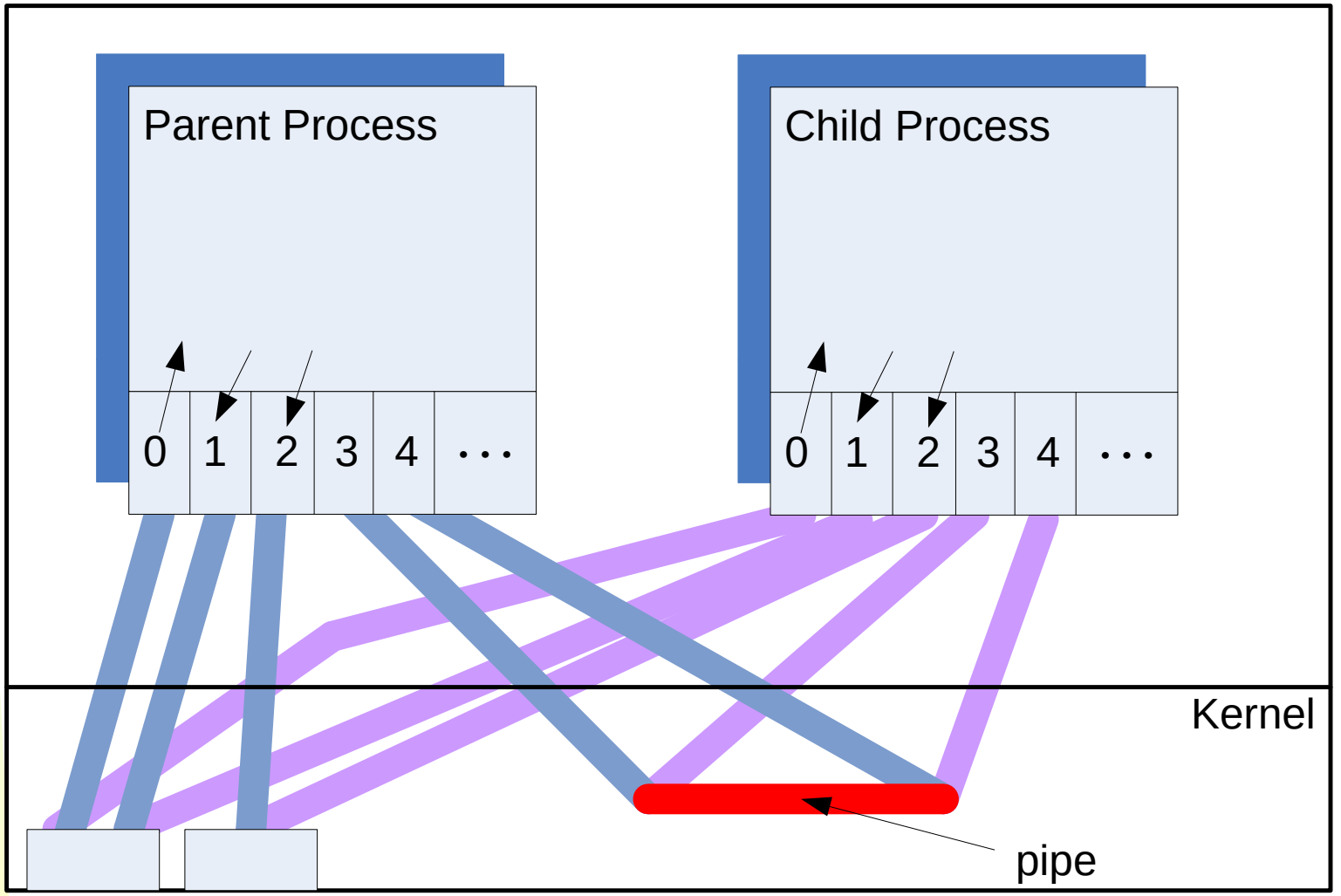
So, how do we make the parent talk to its child:

- **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?**

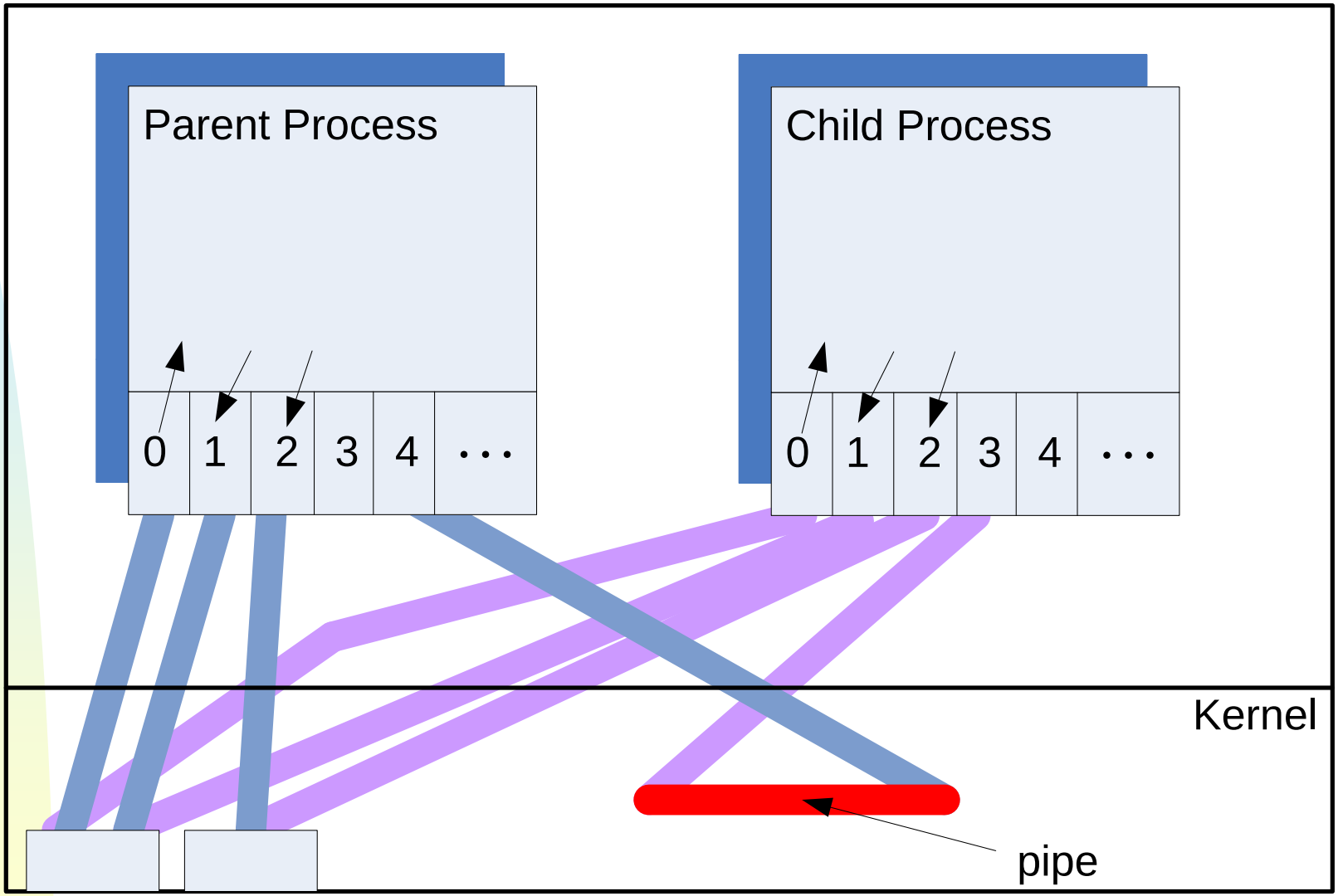
Unix Pipes

```
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

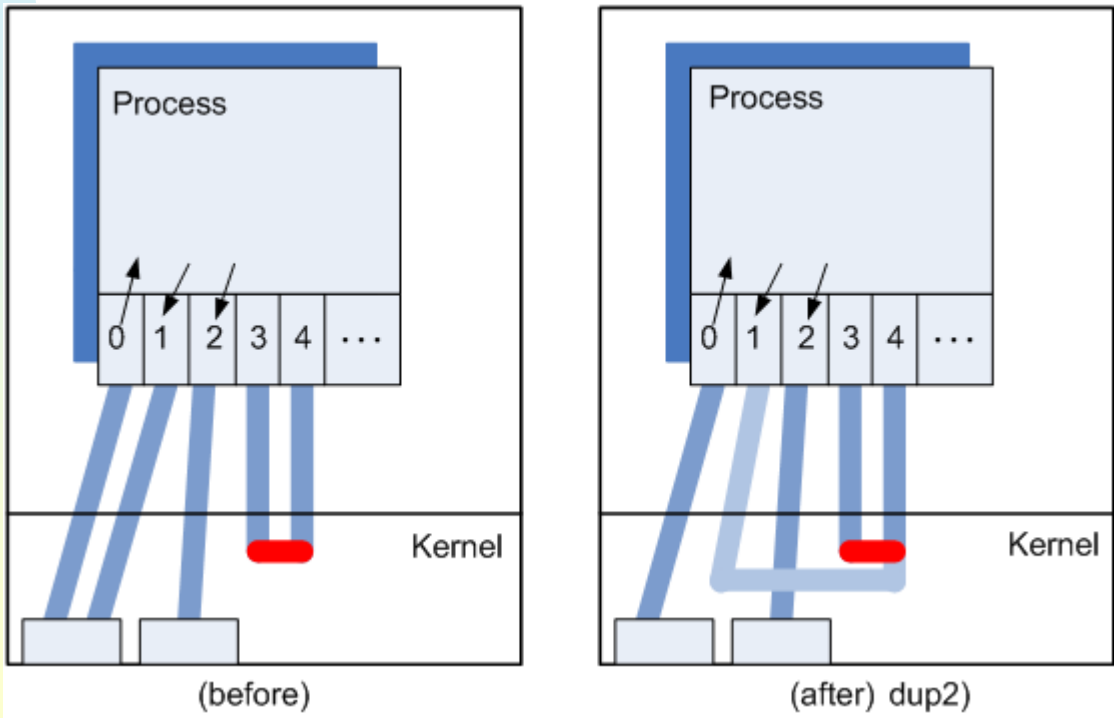


After Closing Unused Ends of Pipe



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)



- **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

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**

Unix Named Pipes

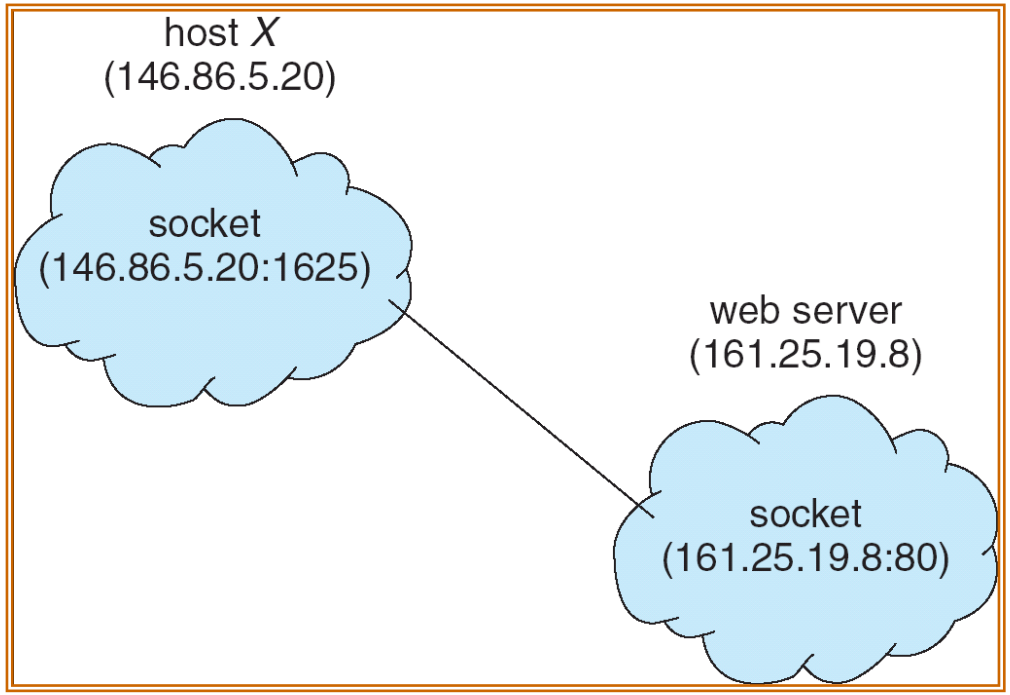
But we can still communicate only between related processes.

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**

Sockets

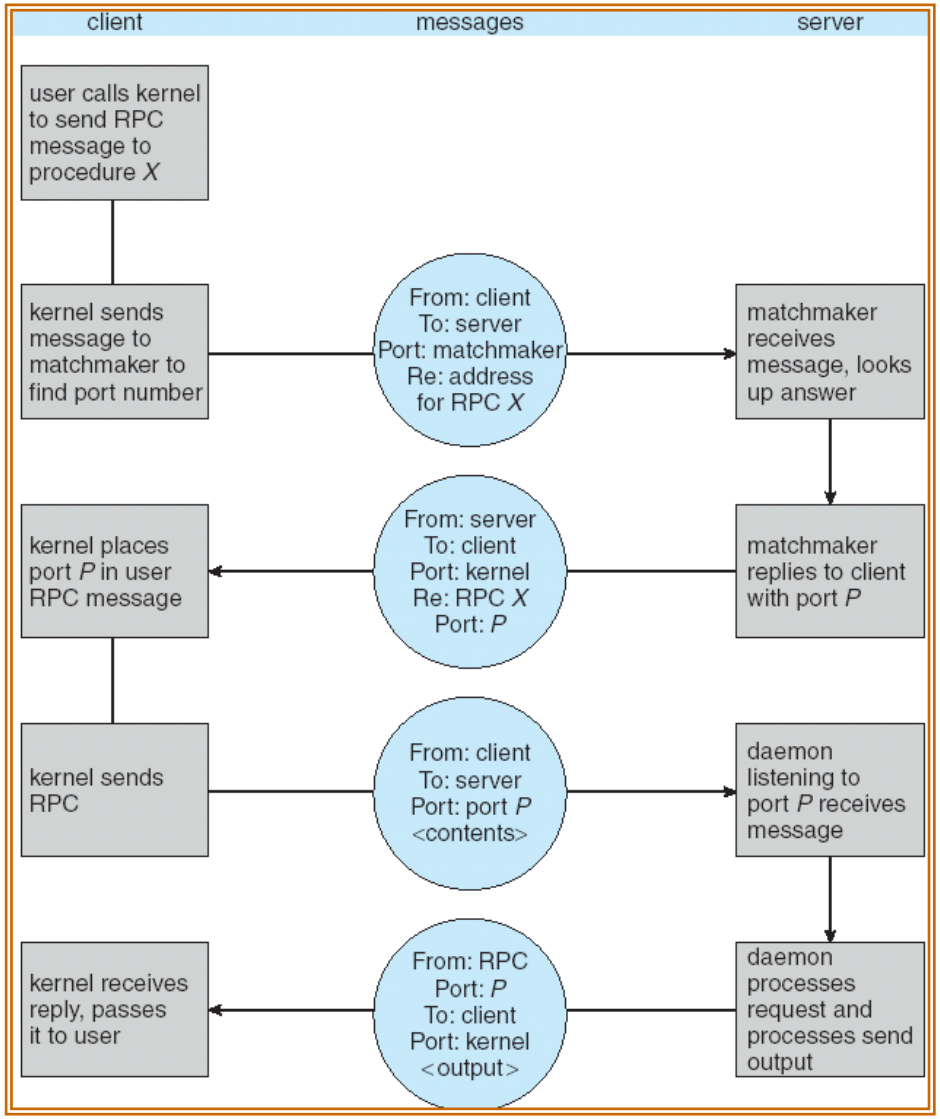
- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication link corresponds to a pair of sockets



Remote Procedure Calls - RPC

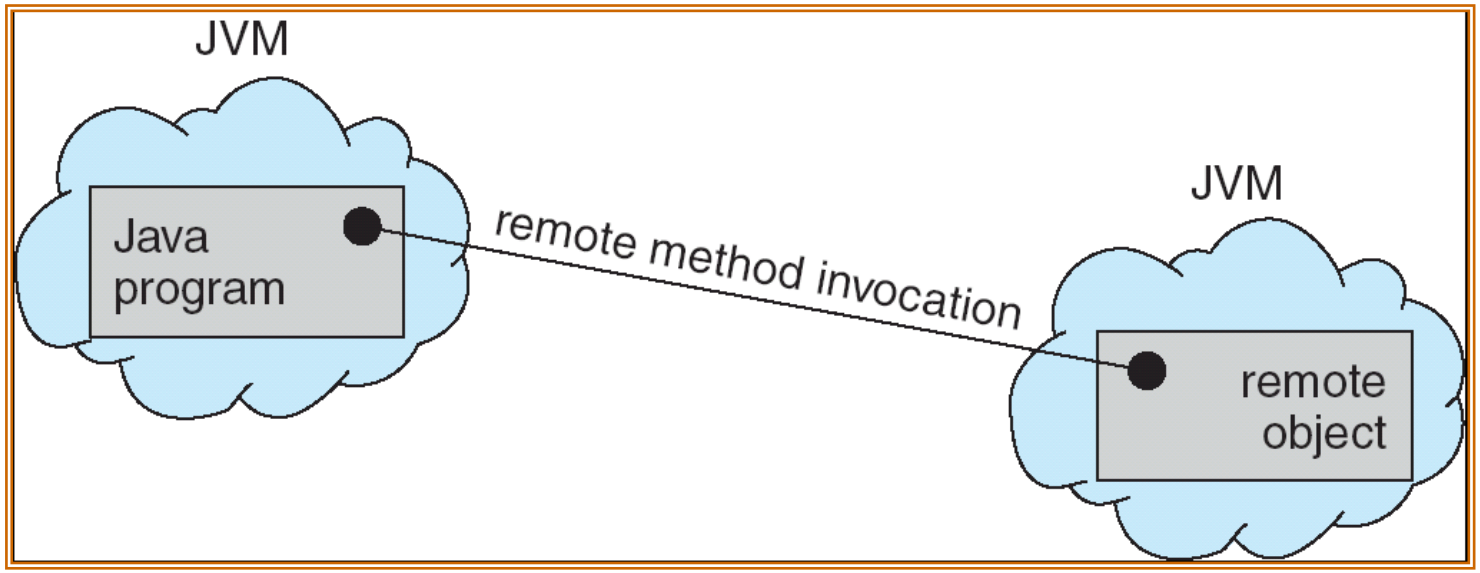
- **Is it possible to make calls to other computers?**
- **Must therefore exchange parameters and return values (possibly multiple data) between computers**
  - **Problem - different computers often have different formats for data representation**
  - **Solution - external representation of data**
- **Support to find the desired server and procedure**
- Replacement Element (Stub): **hides communication details between client and server**
- **At the client, the "stub":**
  - Discover the server
  - Convert parameters (marchalling)
- **At the server, the "stub":**
  - Receive the message and extract the converted parameters
  - Execute the procedure on the server
- **Uses ports to identify servers that can answer various calls - example - file system (NFS from SUN)**

Execution of RPC

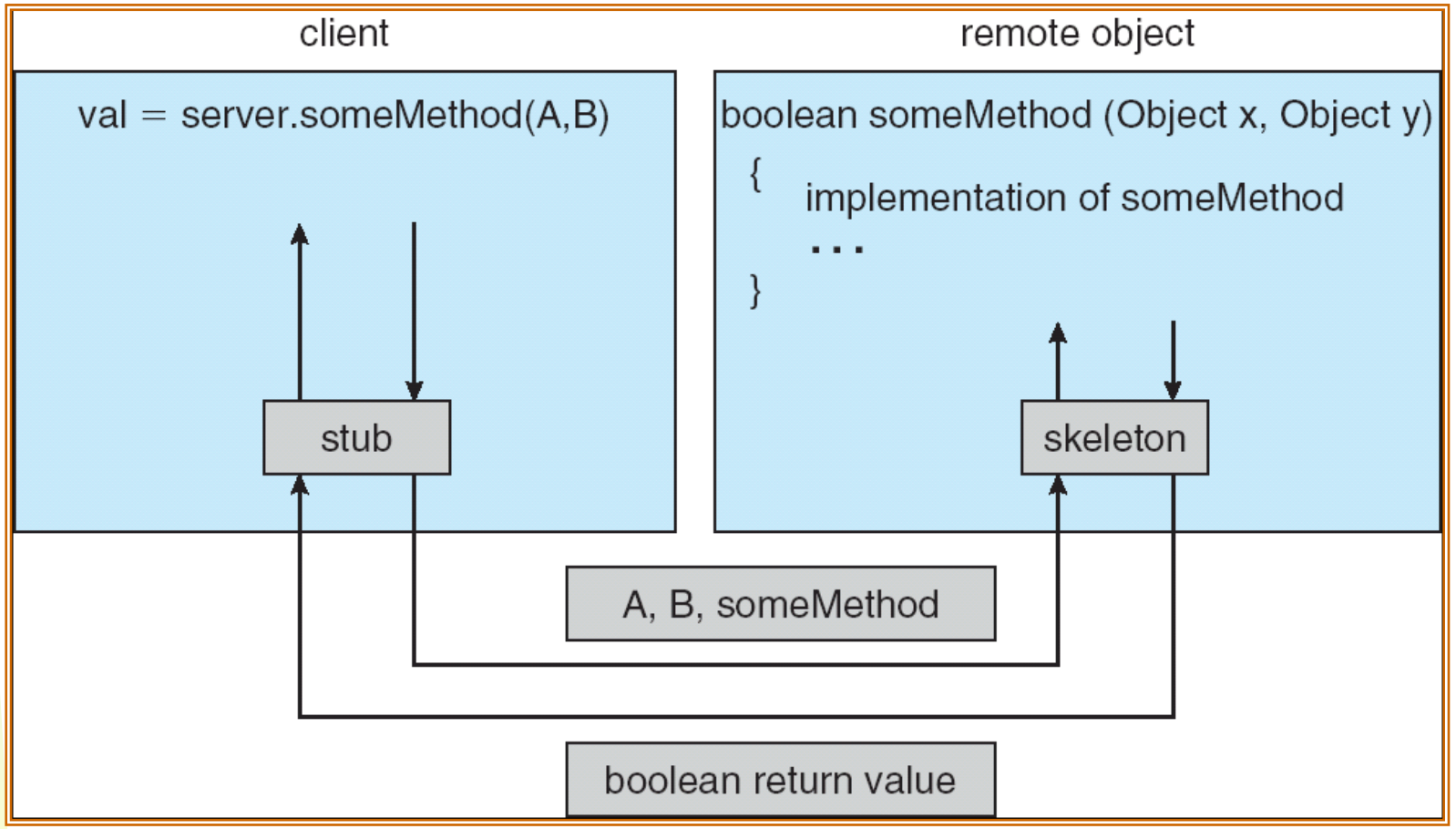


Remote Method Invocation

- 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.



Parameter conversion



- **Local parameters: copy with "object serialization" technique**
- **Remote parameters: use reference**

Important concepts of Module 2

- Process
  - Creation, termination, hierarchy
- States and process state transitions
- Process Control Block PCB
- Process switching
  - PCB saving, reloading
- Process queues and PCBs
- Short, medium and long term schedulers
- Communicating processes
  - IPC communication
  - Pipes, Sockets, RPC, RMI


Thank You!
Ευχαριστώ
ขอบคุณ
Vielen Dank
Merci
Teşekkürler
Dmnvwd
شكراً
متشكرم
Gracias
Grazie
Bedankt
Dankie
THANK YOU
Köszönettel
Obrigado!
شکریا
Díky
謝謝
GADDA GUEY
WAD MAHAD
SAN TAHAY
감사합니다
Urakoze