Process

施吉昇

- Spring 2022
- Daniel Shih/Chung-Wei Lin
- National Taiwan University

國立臺灣大學

NEWS Lab
嵌入式系統暨無線網路實驗室

Outline in OSC

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Outline in this lecture

- Process Concept (ch3@OSC and ch1@xv6)
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Process Concept

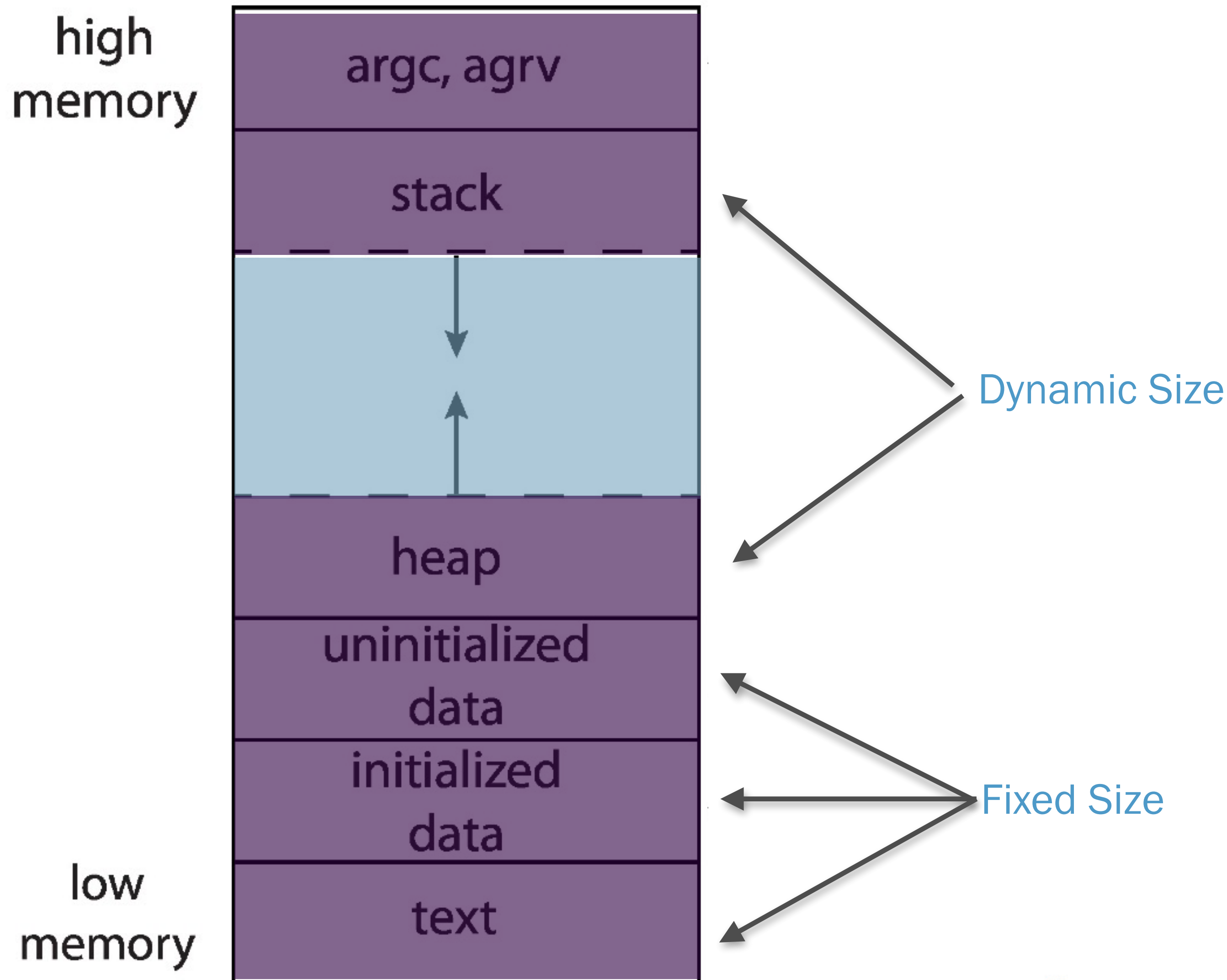
Process Concept

- An operating system executes a variety of programs that run as a process.
- Process – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process.
- Multiple parts
  - The program code, also called text section
  - Current activity including program counter, processor registers
  - Stack containing temporary data
    - Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time

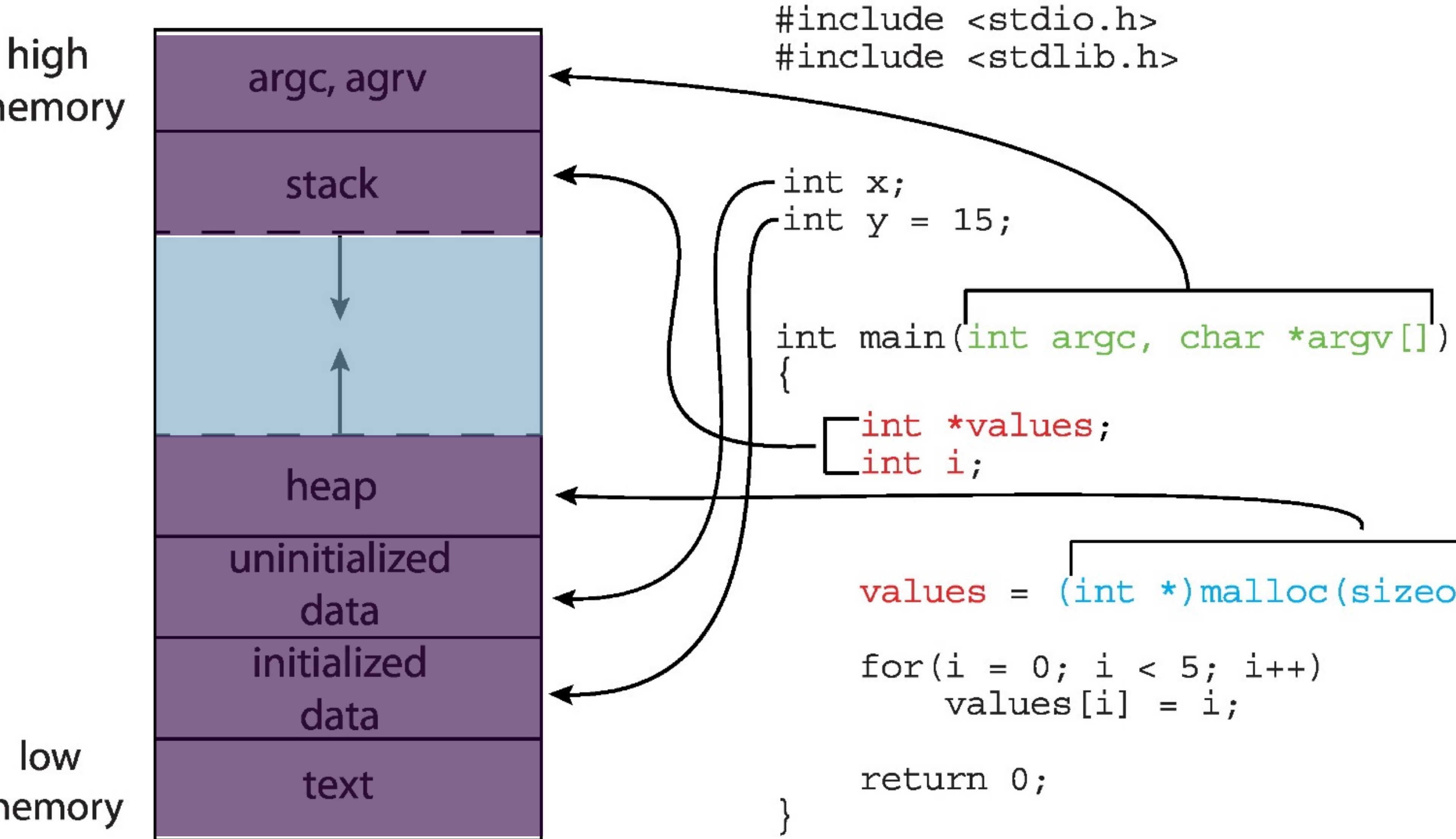
Process Concept (Cont.)

- Program is passive entity stored on disk (executable file); process is active
  - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

Process in Memory



Memory Layout of a C Program



How to check the size of each section?

- Use ‘size’ command to check the size of each section of a *program*:

Linux:

```
[cshih@linux10 ContextSwitch]$ size HelloWorld
   text    data     bss     dec     hex filename
   3179     696       8    3883     f2b HelloWorld
[cshih@linux10 ContextSwitch]$
```

Mac:

```
cshih@MacPro2507 Demo/ContextSwitch %> size HelloWorld
__TEXT  __DATA  __OBJC  others  dec         hex
16384   16384   0       4295000064      4295032832      100010000
cshih@MacPro2507 Demo/ContextSwitch %>
```

```
cshih@MacPro2507 Demo/ContextSwitch %> size -m HelloWorld
Segment __PAGEZERO: 4294967296
Segment __TEXT: 16384
        Section __text: 457
        Section __stubs: 36
        Section __stub_helper: 76
        Section __cstring: 99
        Section __unwind_info: 72
        total 740
Segment __DATA_CONST: 16384
        Section __got: 16
        total 16
Segment __DATA: 16384
        Section __la_symbol_ptr: 48
        Section __data: 8
        total 56
Segment __LINKEDIT: 16384
total 4295032832
cshih@MacPro2507 Demo/ContextSwitch %>
```

How to check the size of each section of a process?

- When a program is loaded and executed, memory, including virtual and physical, will be allocated for different sections of the process.

```
vmmap(1)                    BSD General Commands Manual                    vmmap(1)
NAME
     vmmap -- Display the virtual memory regions allocated in a process
SYNOPSIS
     vmmap [-d seconds] [-w] [-resident] [-pages] [-interleaved] [-submap] [-allSplitLibs] [-
noCoalesce] [-v] pid | partial-executable-name
DESCRIPTION
     vmmap displays the virtual memory regions allocated in a specified process, helping a programmer
understand how memory is being used, and what the purposes of memory at a given address may be.  The
process can be specified by process ID or by full or partial executable name.
OPTIONS
     -d seconds     Take two snapshots of the vm regions of the process, separated by the specified
time, and print the delta between those snap-
                    shots.
     -w, -wide       Print wide output.
     -resident       Show both the virtual and resident sizes for each region, in the form [ virtual/
resident].

     -pages     Print region sizes in page counts rather than kilobytes.
```

Process State

Requirements of running processes in OS

Multiplexing/Time-Sharing

Isolation

Interaction

Requirements of running processes in OS

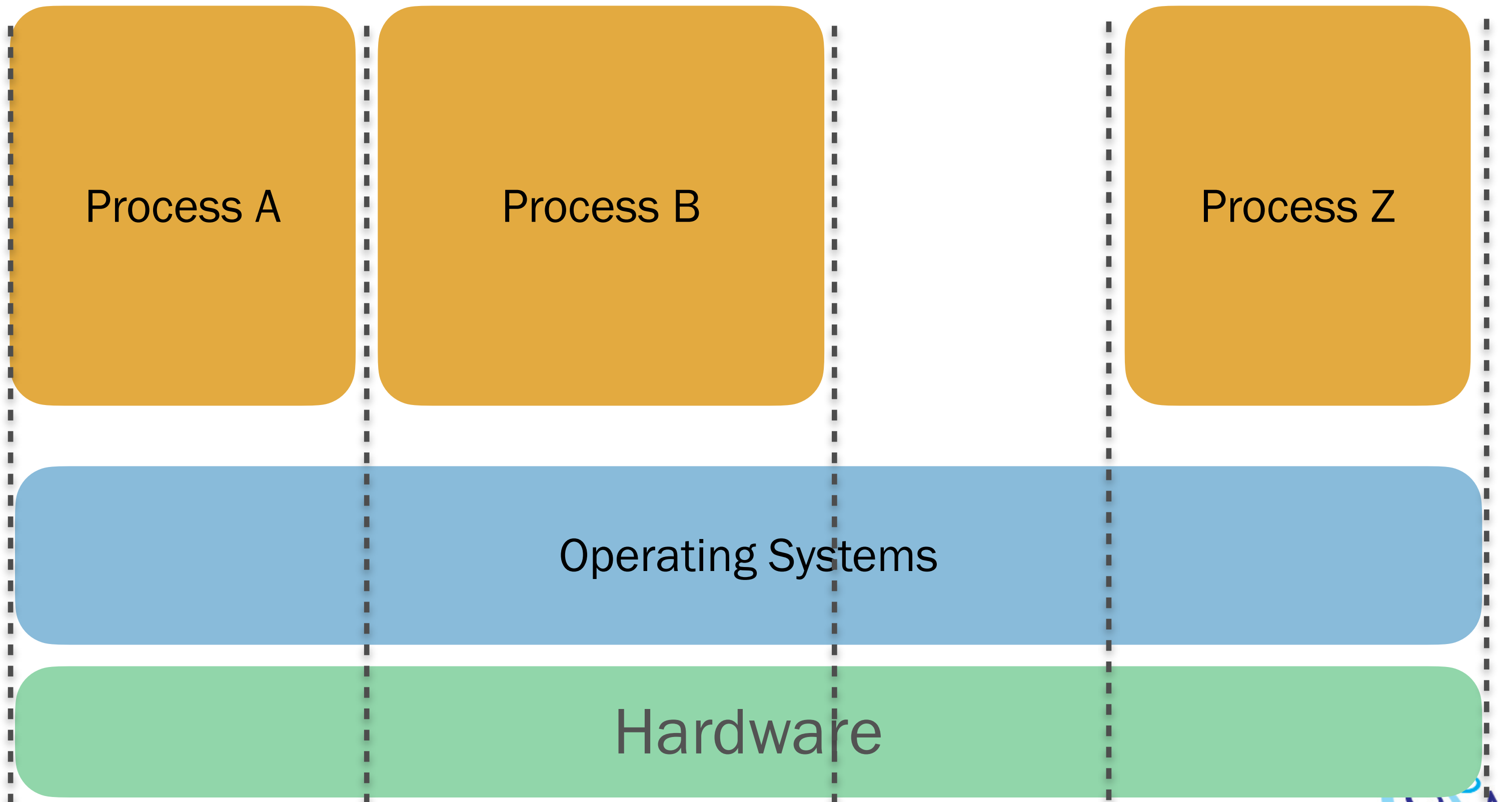
Multiplexing/ Time-Sharing

on modern computers, physical and logical resources need to be shared among numbers of processes. OS needs to allow the processes to occupy AND yield the resources from time to time (at sub-second level.)

Requirements of running processes in OS

Isolation

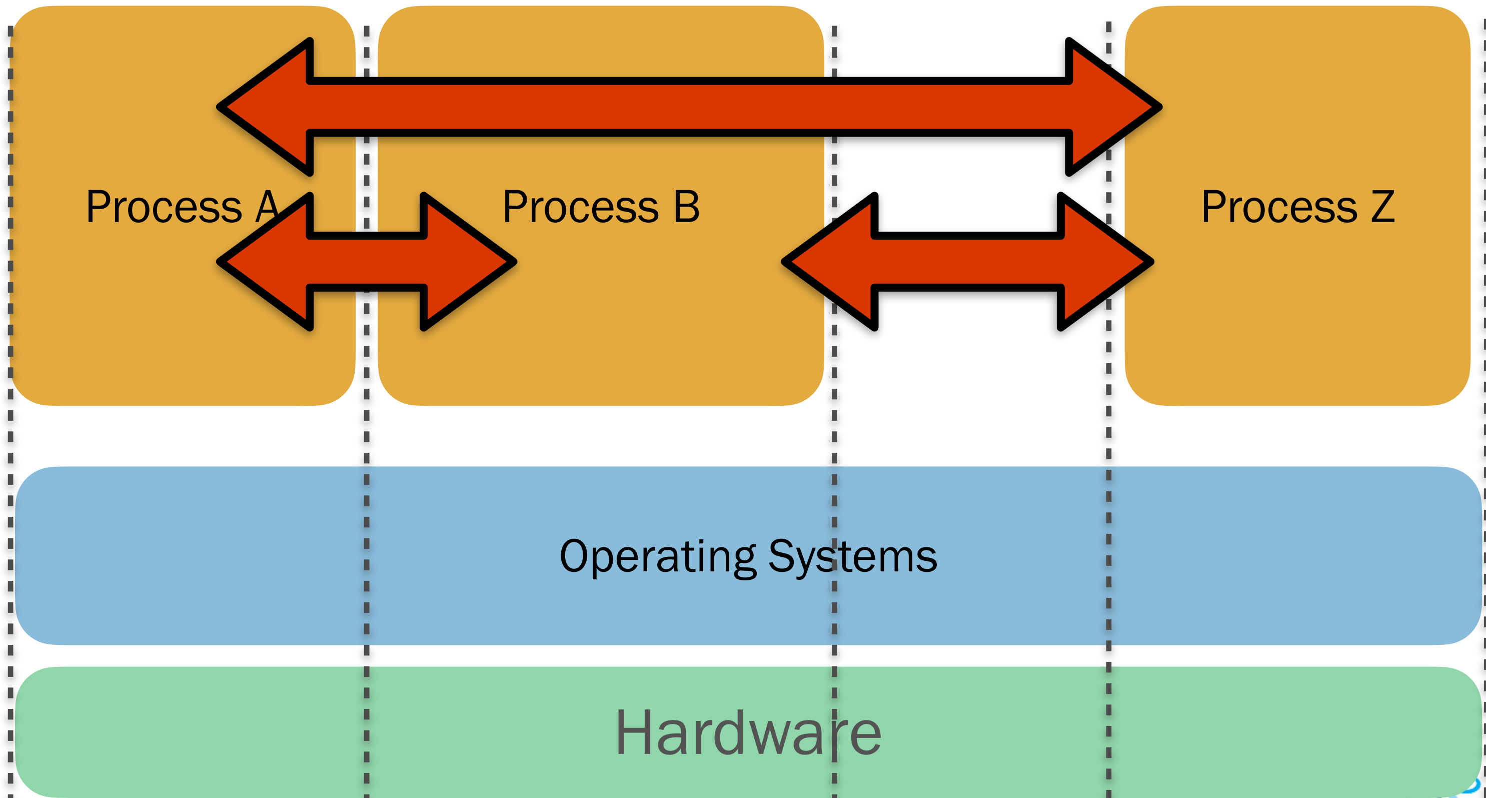
the operating system should protect each process being abused or attacked on resource assignment, data storage, and execution.



Requirements of running processes in OS

Interaction

resources, including data and code, should be shared among process.



Requirements of running processes in OS

Multiplexing/ Time-Sharing

on modern computers, physical and logical resources need to be shared among numbers of processes. OS needs to allow the processes to occupy AND yield the resources from time to time (at sub-second level.)

Isolation

the operating system should protect each process being abused or attacked on resource assignment, data storage, and execution.

Interaction

resources, including data and code, should be shared among process.

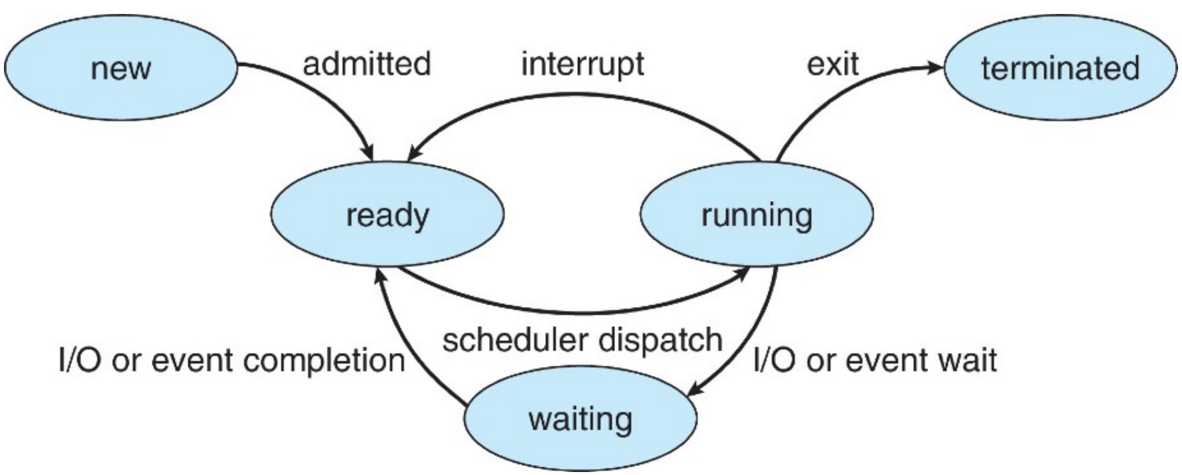
How to meet these requirements

- The processes in the same operating system need to share
  - the processors but one process at a time,
  - the memory, and
  - other peripheral devices.
- On processors, the OS schedules the processes to execute:
  - One at a time,
  - One process can only be scheduled to execute on processor at certain cases and to suspend from processor at certain cases.
- Process State is the way to allow the OS to decide when to schedule properly.

NEWS Lab
嵌入式系統暨無線網路實驗室

Process State

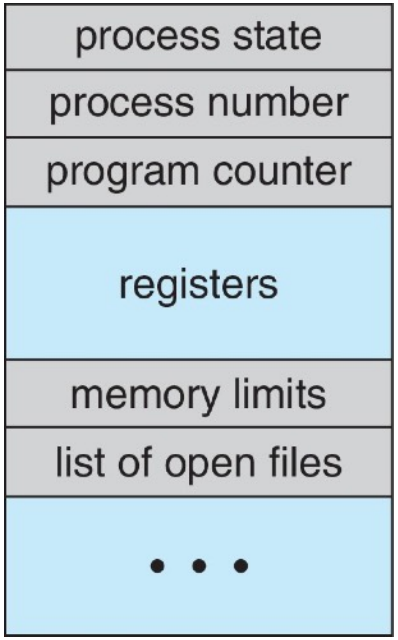
- As a process executes, it changes its **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution



Process Control Block (PCB)

Information associated with each process (also called task control block)

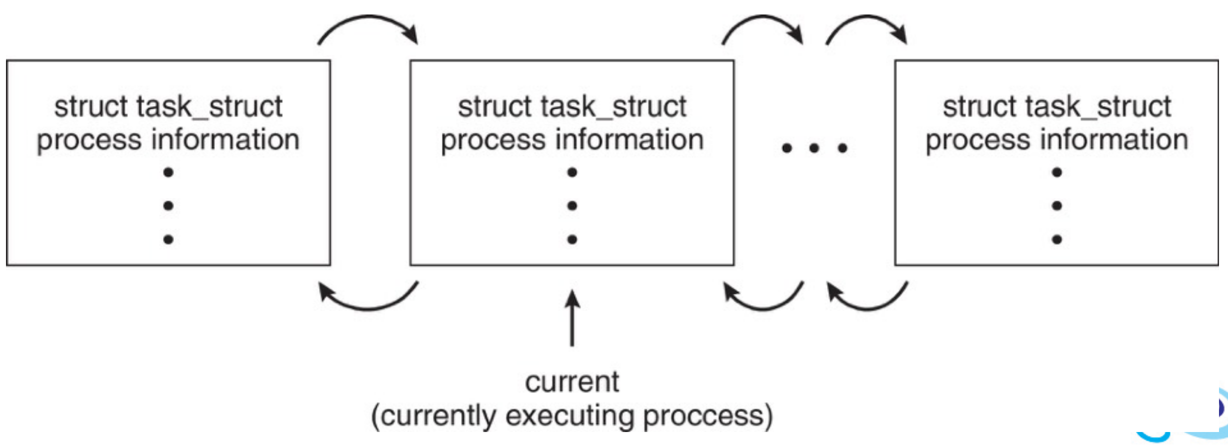
- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting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



Process Representation in Linux

Represented by the C structure `task_struct`

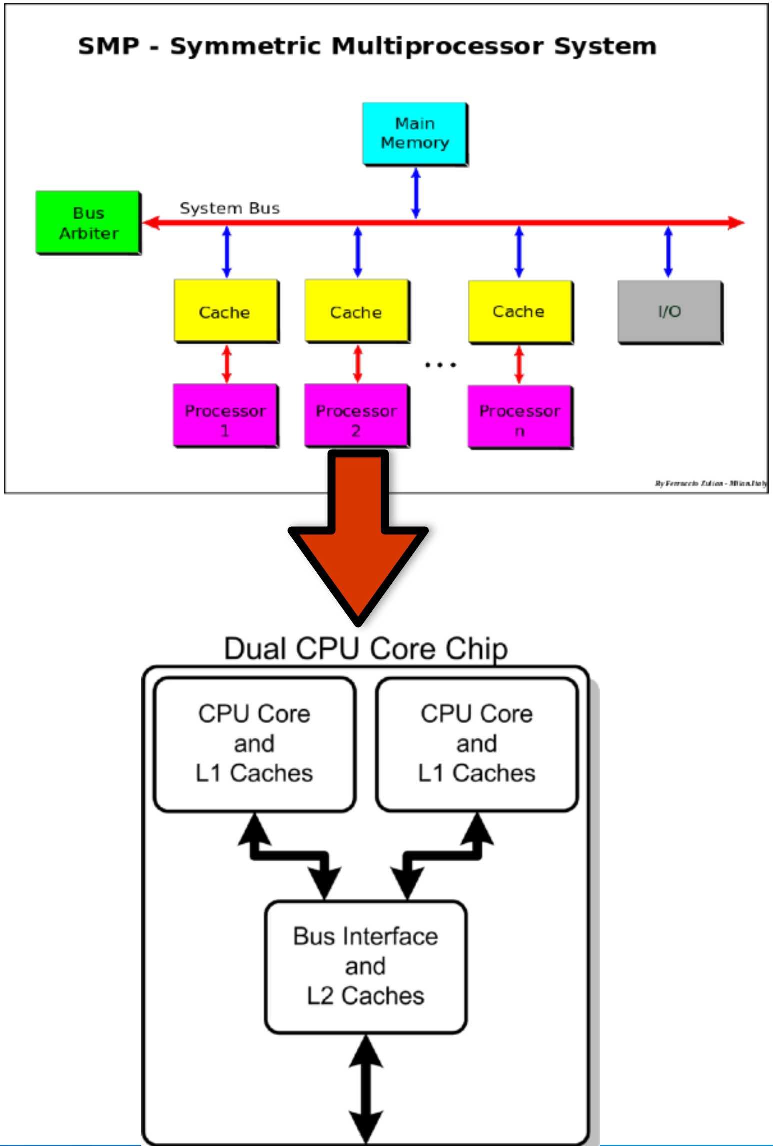
```
pid t_pid;                    /* process identifier */
long state;                   /* state of the process */
unsigned int time_slice       /* scheduling information */
struct task_struct *parent;/* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;         /* address space of this process */
```



How about Multi-core or Multi-processor?

NEWS Lab
嵌入式系統暨無線網路實驗室

Multi-processor vs. Multi-core



- Each processor has its own cache but share main memory and IO with other processors via shared Bus.

- Within each processor, there are more than one processing unit, called a core.
- Each core has its own L1 cache but may share other cache with other cores of same processor.

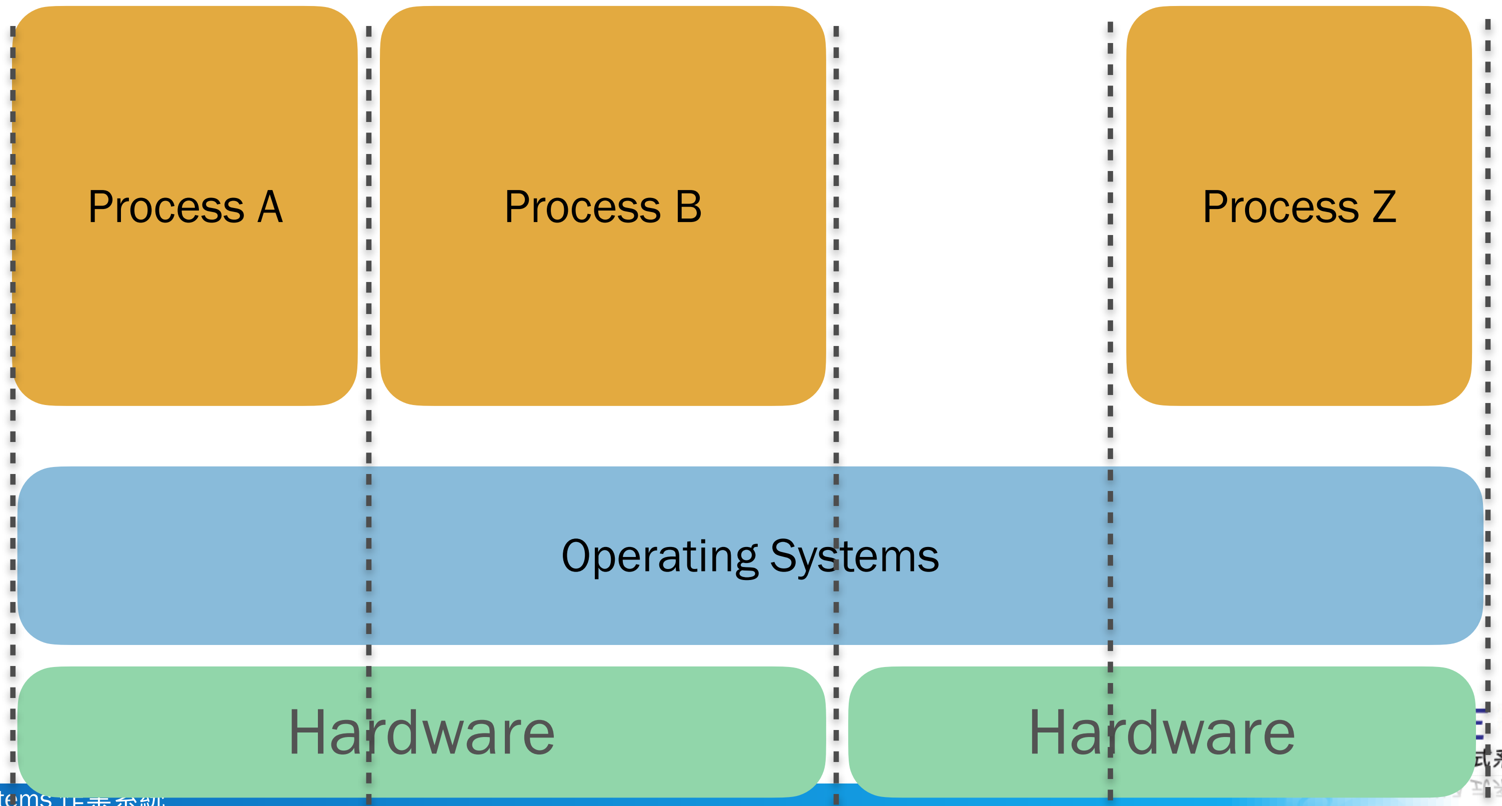
How to meet these requirements

- The processes in the same operating systems need to share
  - the processors but one process at a time,
  - the memory, and
  - other peripheral devices.
- On processor, the OS schedules the processes to execute:
  - One at a time,
  - One process can only be scheduled to execute on processor at certain cases and to suspend from processor at certain cases.
- Process State is the way to allow the OS to decide when to schedule properly.

How about running processes on different processors?

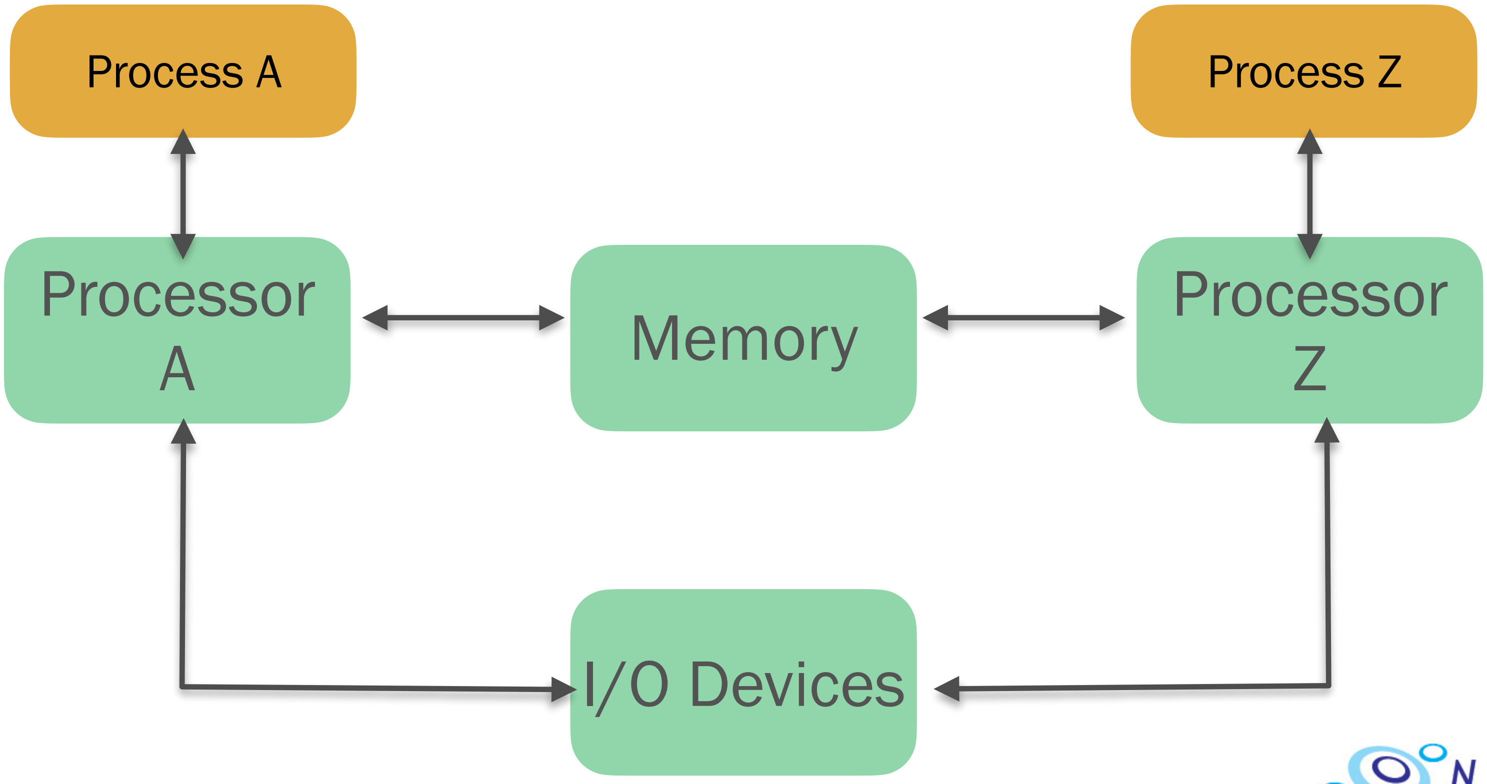
When there are multiple processors

- Can the OS schedule different processes to run on different processors?
- Technically, YES. But, the OS needs to re-design for multiple-processors systems, which is one type of distributed systems.
  - There are multiple processors. How about memory, disk, IO devices, etc?



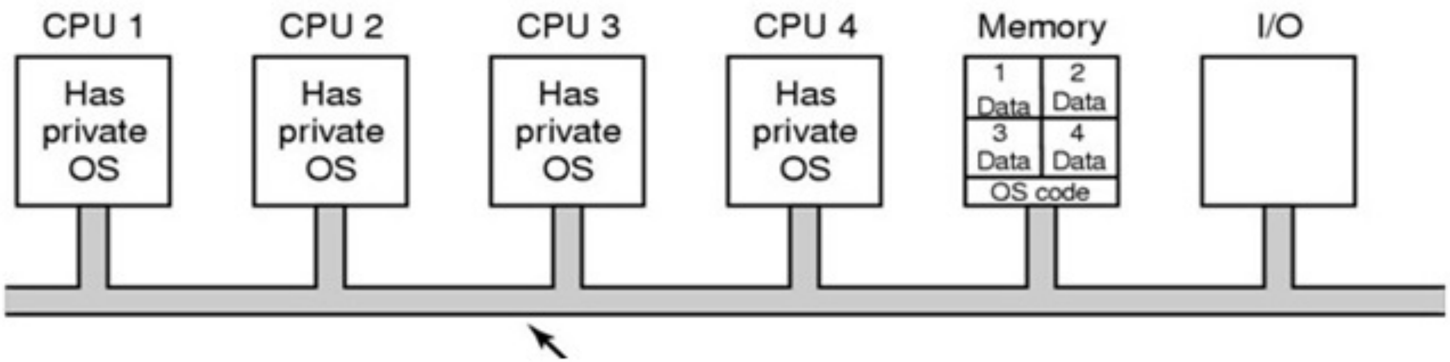
When there are multiple processors

- Can the OS schedule different processes to run on different processors?
- Technically, YES. But, the OS needs to re-design for multiple-processors systems, which is one type of distributed systems.
  - There are multiple processors. How about memory, disk, IO devices, etc?



Multiple Processor OS

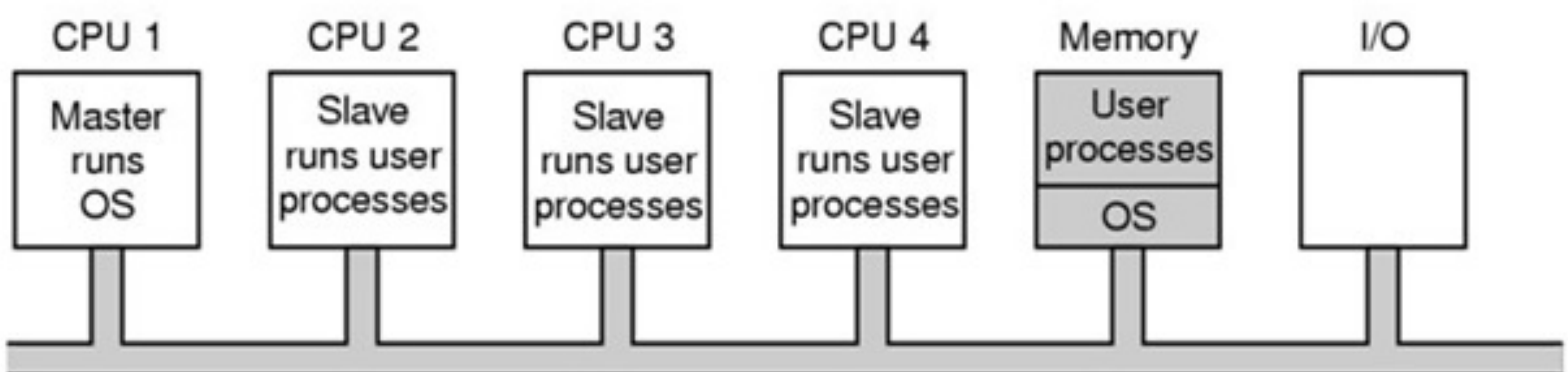
- Each CPU has its own private operating system and memory is shared among all the processors and input-output system are also shared. All the system is connected by the single bus.



Multiple Processor OS

Master slave multiprocessor

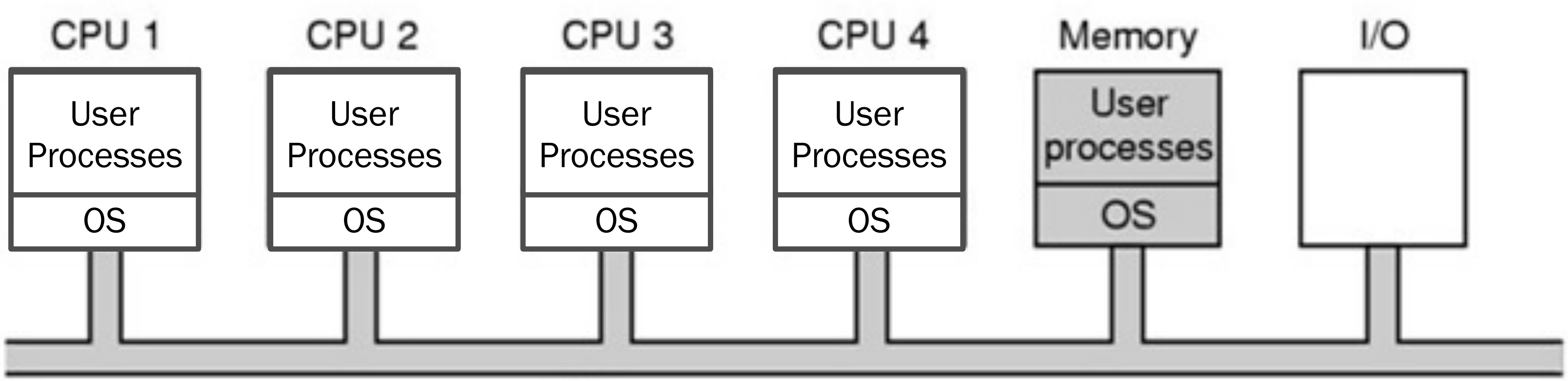
- There is a single data structure which keeps track of the ready processes.
- In this model, one processor works as master and other central processing unit work as a slave.
  - All the processors are managed by the single processor which is called master server.
  - The master server runs the operating system process and the slave server run the user processes.
  - The memory and input-output devices are shared among all the processors and all the processor are connected to a common bus.
- This system is simple and reduces the data sharing so this system is called *Asymmetric multiprocessing*.



Multiple Processor OS

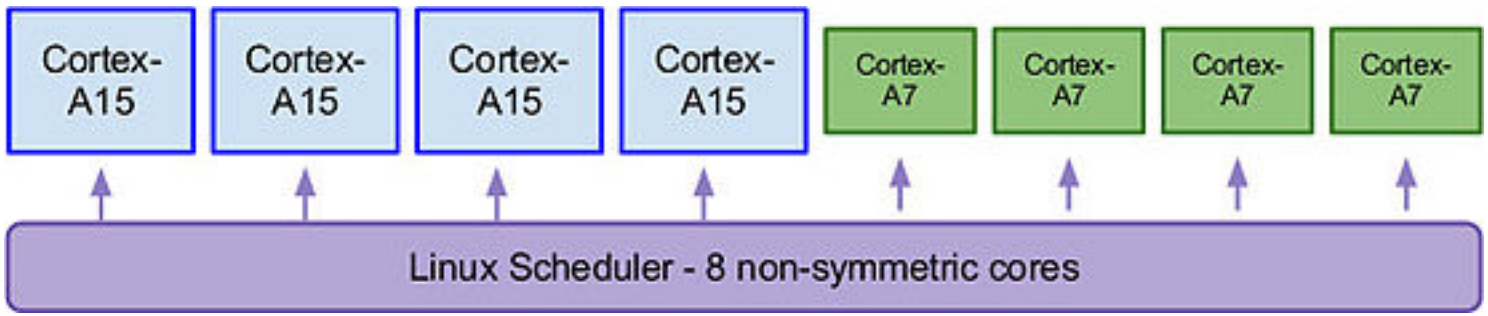
Symmetric multiprocessor (SMP)

- In SMP model, there is one copy of the OS in memory, but any processor can run it.
- When a system call is made, the processor on which the system call was made traps to the kernel and then processes that system call.
  - This model balances processes and memory dynamical.
- This approach uses Symmetric Multiprocessing where each processor is self-scheduling. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.



Symmetric or not?

- Questions: Are the processors required to be symmetric?



The most powerful use model of big.LITTLE architecture is Heterogeneous Multi-Processing (HMP), which enables the use of all physical cores at the same time. Threads with high priority or computational intensity can in this case be allocated to the "big" cores while threads with less priority or less computational intensity, such as background tasks, can be performed by the "LITTLE" cores.

This model has been implemented in the Samsung Exynos starting with the Exynos 5 Octa series (5420, 5422, 5430), and Apple A series processors starting with the Apple A11.

Process Scheduling

NEWS Lab
嵌入式系統暨無線網路實驗室

Process Scheduling

- Process scheduler selects among available processes for next execution on CPU core.
  - For a system with a single core, there will never be more than one process CPU running at a time, whereas a multicore system can run multiple processes at one time.
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains scheduling queues of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
- Processes migrate among the various queues

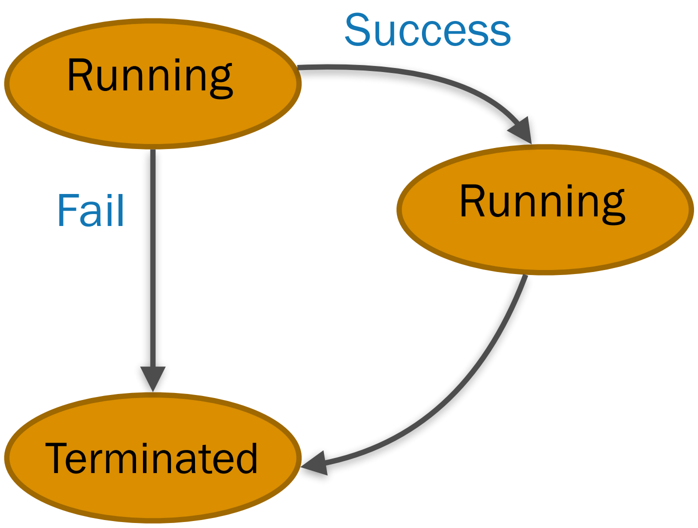
Not Every Process Has Same Workload

- Optimally assigning processor time is not trivial because not all processes have the same workload pattern.
  - An **I/O-bound** process is one that spends more of its time doing I/O than it spends doing computations.
  - A **CPU-bound** process, in contrast, generates I/O requests infrequently, using more of its time doing computations.
- In general, CPU-bound process should receive more time to execute. However, some process/programmer may cheat to acquire more process time.
  - **Spin lock** is an interactive loop to wait on certain resources, usually IO, but does not release processing time.
  - It should be used carefully.

How to Lock?

- Lock a resource: Once and for good
  - The process tries to lock the resource.
  - If not available, it may abort or come back later.

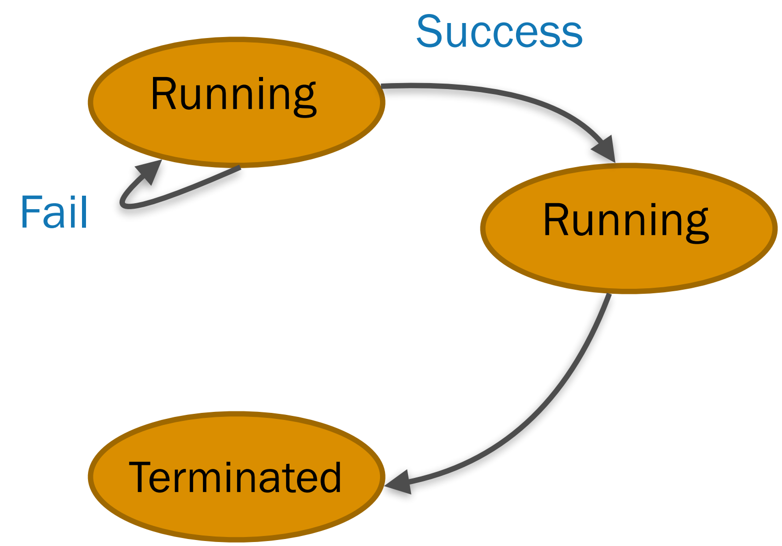
```
if lock(myFile) {
	printf("this is what I want to write.\n");
	unlock (myFile);
}
```



How to Lock?

- Spin Lock: Keep trying
  - The process repeatedly tries to lock the resource.
  - During the trials, the process stays in running state and consume CPU time.

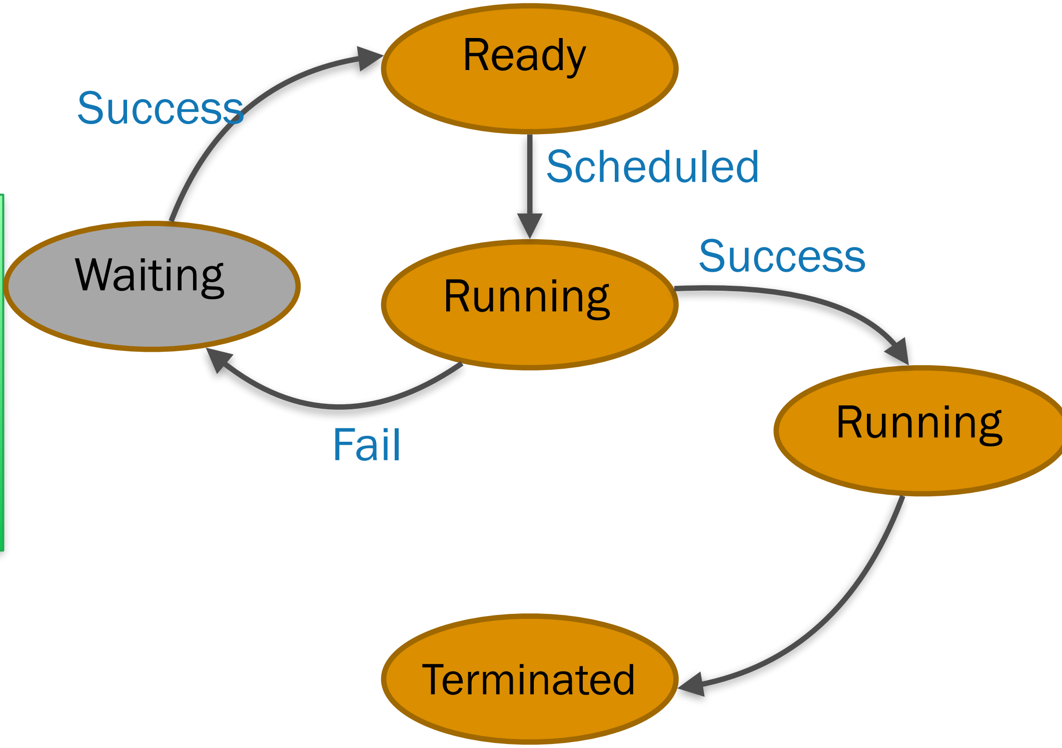
```
while  {not lock(myFile)} { };

printf("this is what I want to write.\n");

unlock (myFile);
```



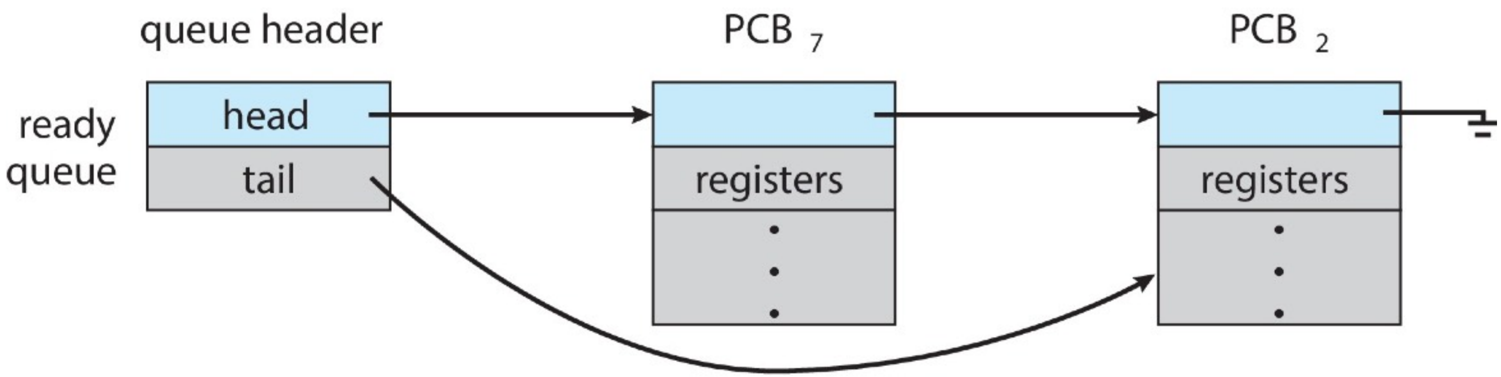
How to Lock?

- Mutex Lock: wait for lock
  - The process waits for the signal when the resource is available.
  - During the waiting, the process is suspended.

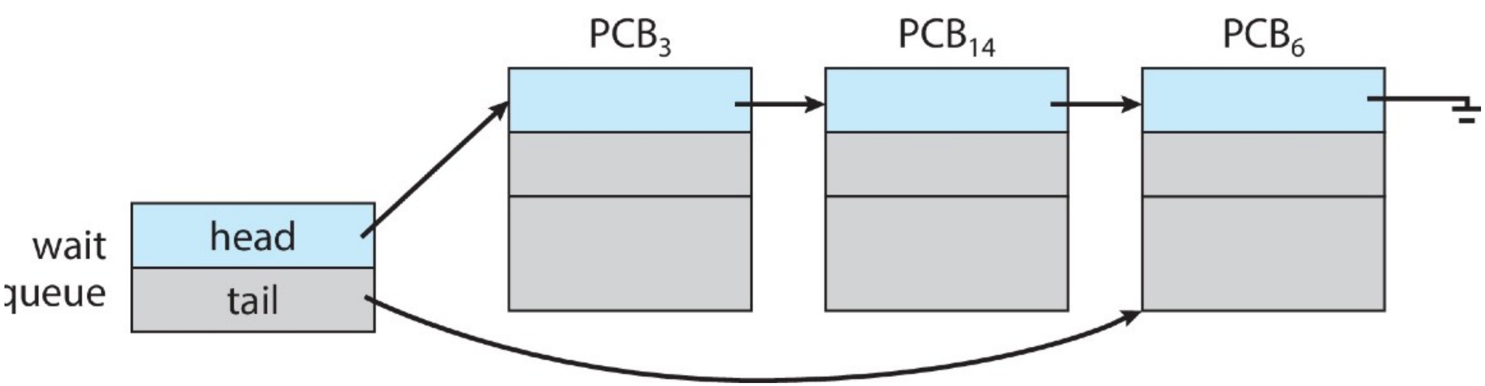
```
if acquire(myFile.lock) {
      printf("this is what I want to write.
      \n");
      release (myFile.lock);
}
```



Ready and Wait Queues

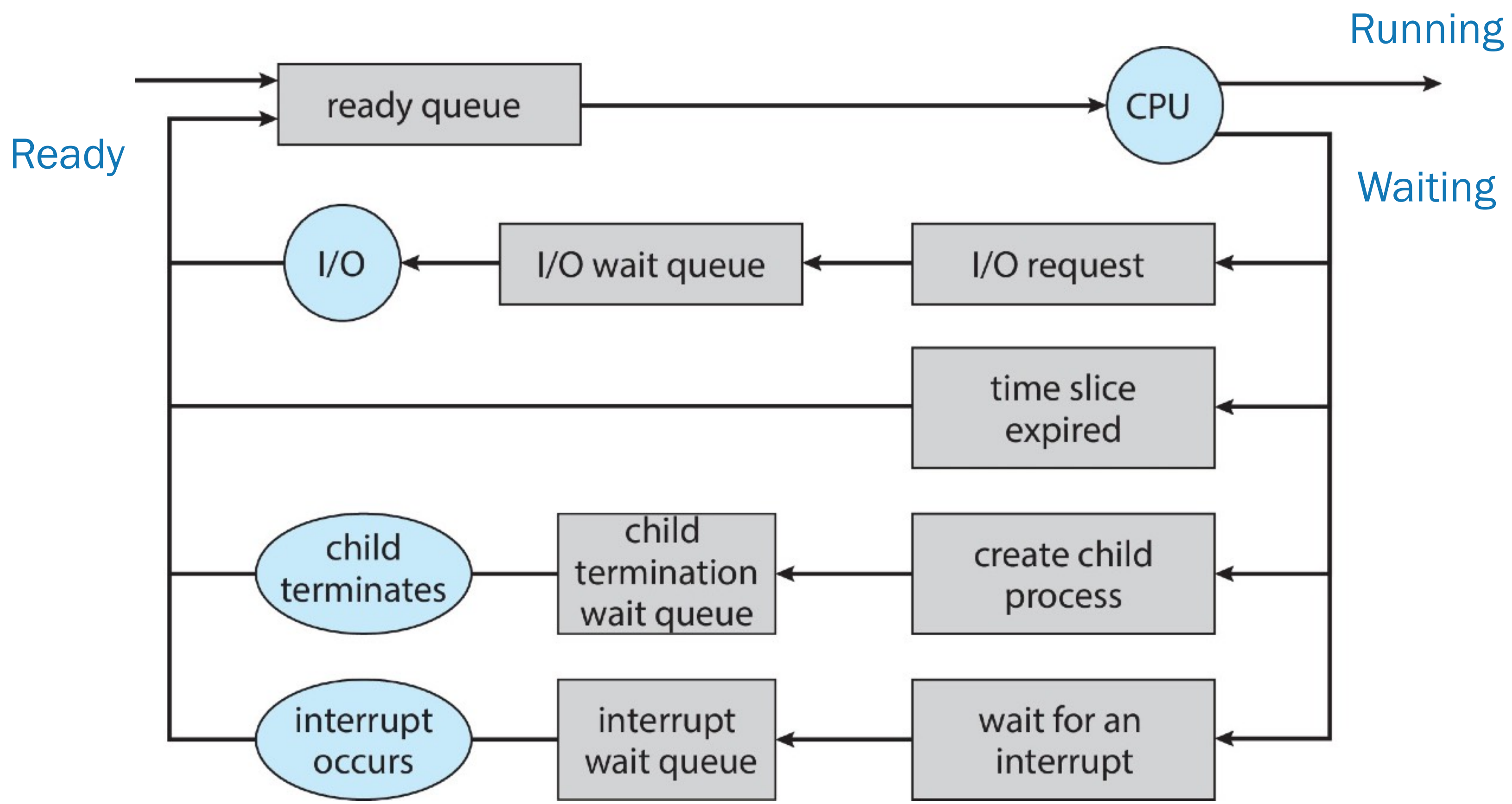


Ready Queue



Wait Queue

Representation of Process Scheduling



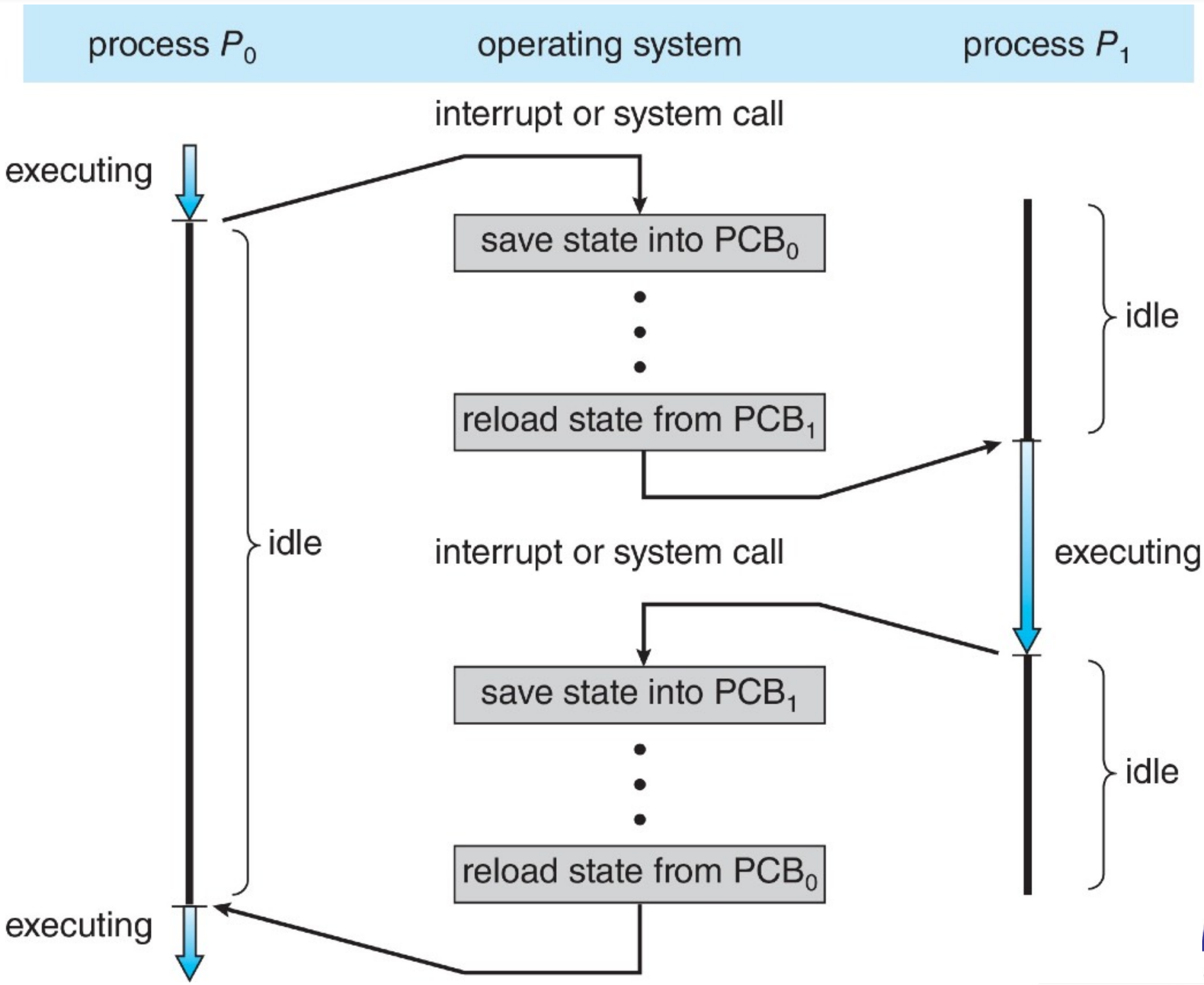
Queueing-diagram representation of process scheduling.

Context Switch

- 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 via a context switch
- Context of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch

CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.



Context Switch

- 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 via a context switch
- Context of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

When to switch?

Context switch can occur voluntarily or involuntarily

- Voluntarily context switch: one process explicitly yields the CPU to another.
  - Call yield() to yield CPU
  - Call sleep()
  - Request I/O, request to lock/semaphore, makes a system call that blocks, etc.
- Involuntarily context switch: the system scheduler suspends an active process, and switches control to a different process.
  - Process/thread scheduler tries to share CPU fairly by time-slicing.
  - Suspend/resume at periodic intervals
  - Involuntary context switches can happen “any time”.

NEWS Lab
嵌入式系統暨無線網路實驗室

Demo Context Switch

- **I/O-bound** process: lead to voluntary and involuntary context switch

```
cshih@MacPro2507 Demo/ContextSwitch %> ./HelloWorld -i | more        [20:57:00]
Hello World, OS Spring 2022.
Hello World, OS Spring 2022.
Hello World, OS Spring 2022.
Hello World, OS Spring 2022.
Hello World, OS Spring 2022.
Hello World, OS Spring 2022.
```

```
Hello World, OS Spring 2022.
Hello World, OS Spring 2022.
Voluntary Context Switch: 289
Involuntary Context Switch: 107
396 context switches in 196738000 ns (496813.1ns / switch)
```

- **CPU-bound** process: only has involuntary context switch or no context switch

```
cshih@MacPro2507 Demo/ContextSwitch %> ./HelloWorld                   [20:53:43]
Hello World, OS Spring 2022.
Voluntary Context Switch: 0
Involuntary Context Switch: 3
3 context switches in 344000 ns (114666.7ns / switch)
```

Cost of Context Switch

- Direct Cost: Load and Store instructions

```
cshih@MacPro2507 Demo/ContextSwitch %> ./HelloWorld                [20:53:43]
Hello World, OS Spring 2022.
Voluntary Context Switch: 0
Involuntary Context Switch: 3
3 context switches in 344000 ns (114666.7ns / switch)
```

- Indirect cost: COLD cache and cache misses
  - HOT Cache: the required data are stored in cache. No need to fetch data/ instruction from main memory.
  - COLD Cache: just a blank cache or one with stale-data. Need to re-fetch data/ instructions from main memory.

NEWS Lab
嵌入式系統暨無線網路實驗室

Different behavior on sleep() for CS

- nanosleep() on osx leads to involuntary context switch

```
cshih@MacPro2507 Demo/ContextSwitch %> ./HelloWorld -s                [20:57:39]
Hello World, OS Spring 2022.
Voluntary Context Switch: 0
Involuntary Context Switch: 10081
10081 context switches in 171864000 ns (17048.3ns / switch)
```

```
[cshih@linux10 ContextSwitch]$ ./HelloWorld -s
Hello World, OS Spring 2021.
Voluntary Context Switch: 99995
Involuntary Context Switch: 2
99997 context switches in 6031674173 ns (60318.6ns / switch)
```

- nanosleep(), which suspends the process for very short period of time, may be implemented by a spin-lock and keeps the processor busy. The process will be rescheduled *involuntarily* when the time slice expires.
- sleep() puts the process into waiting state and waiting queue, which causes a *voluntary* context switch.

Communications in Client-Server Systems

NEWS Lab
嵌入式系統暨無線網路實驗室

Communications

Communication in Single Computer

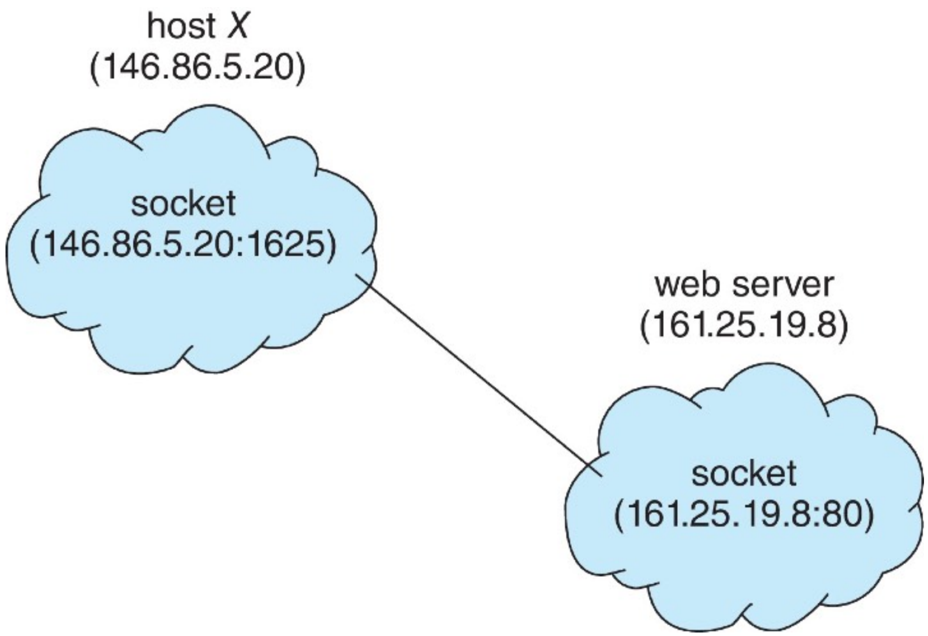
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems

Communication in Multiple Computers

- Communication in Client-Server Systems
  - Sockets
  - Remote Procedure Calls

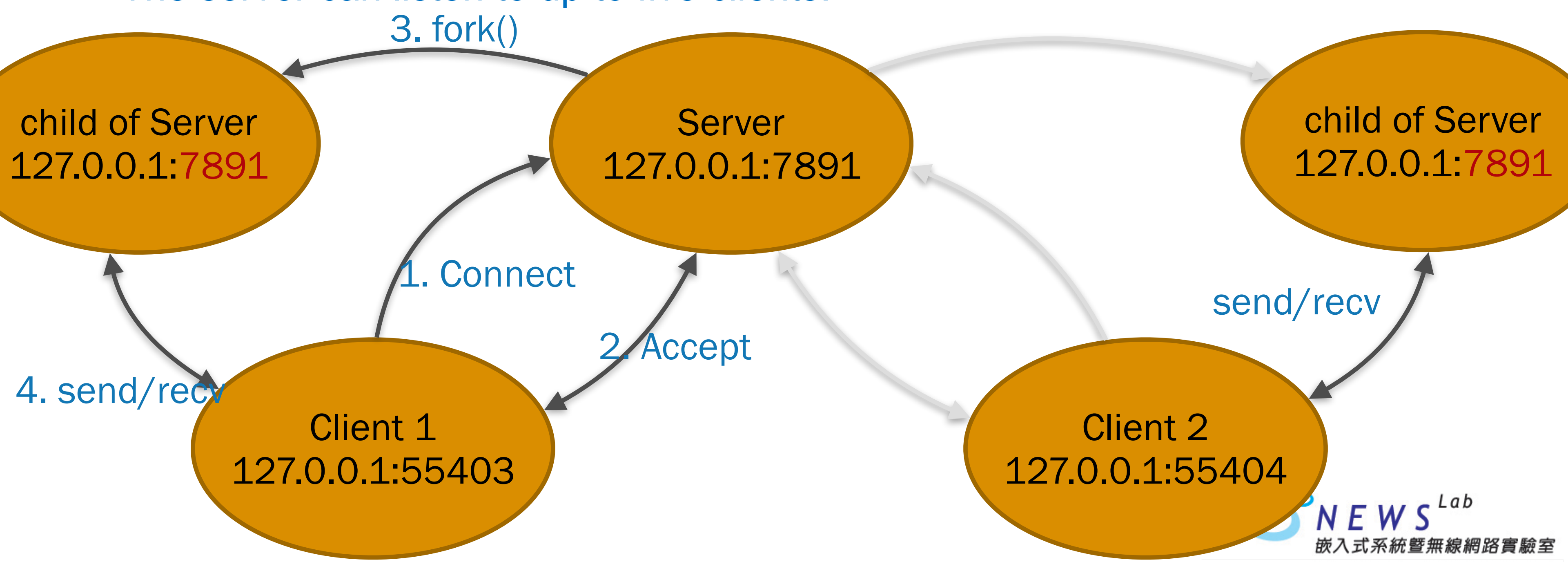
Sockets on different machines

- A socket is defined as an endpoint for communication
- Concatenation of IP address and port – a number included at start of message packet to differentiate network services on a host
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8



Socket Example: Client-Server

- Client and server are located on the same machine: 127.0.0.1
- The port number is given to both client and server and fixed. (Not good)
- The Server listens to the string from client and changes the received string into upper case and send back.
- The server can listen to up to five clients.



Socket Example: Server

```c
int main(){
  int welcomeSocket, newSocket, portNum, clientLen, nBytes;
  char buffer[1024];
  struct sockaddr_in serverAddr;
  struct sockaddr_storage serverStorage;
  socklen_t addr_size;
  int i;

  welcomeSocket = socket(PF_INET, SOCK_STREAM, 0);

  portNum = 7891;

  memset(&serverAddr, '\0', sizeof(serverAddr));
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_port = htons(portNum);
  serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
  memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);

  bind(welcomeSocket, (struct sockaddr *) &serverAddr, sizeof(serverAddr));

  if(listen(welcomeSocket,5)==0)
    printf("Listening\n");
  else
    printf("Error\n");

  addr_size = sizeof serverStorage;

```

Socket Example: Server

```c
  /*loop to keep accepting new connections*/
  while(1){
    newSocket = accept(welcomeSocket, (struct sockaddr *) &serverStorage, &addr_size);

    /*fork a child process to handle the new connection*/
    if(!fork()){
  struct sockaddr_in childAddr;

  memset(&childAddr, '\0', sizeof(childAddr));
  int len = sizeof(childAddr);
  nBytes = 1;
  /*loop while connection is live*/
  while(nBytes!=0){
      nBytes = recv(newSocket,buffer,1024,0);

      for (i=0;i<nBytes-1;i++){
       buffer[i] = toupper(buffer[i]);
      }
      send(newSocket,buffer,nBytes,0);
  }
  printf("Close one socket.\n");
  close(newSocket);
  exit(0);
    }
    /*if parent, close the socket and go back to listening new requests*/
    else{
      close(newSocket);
    }
  }
```

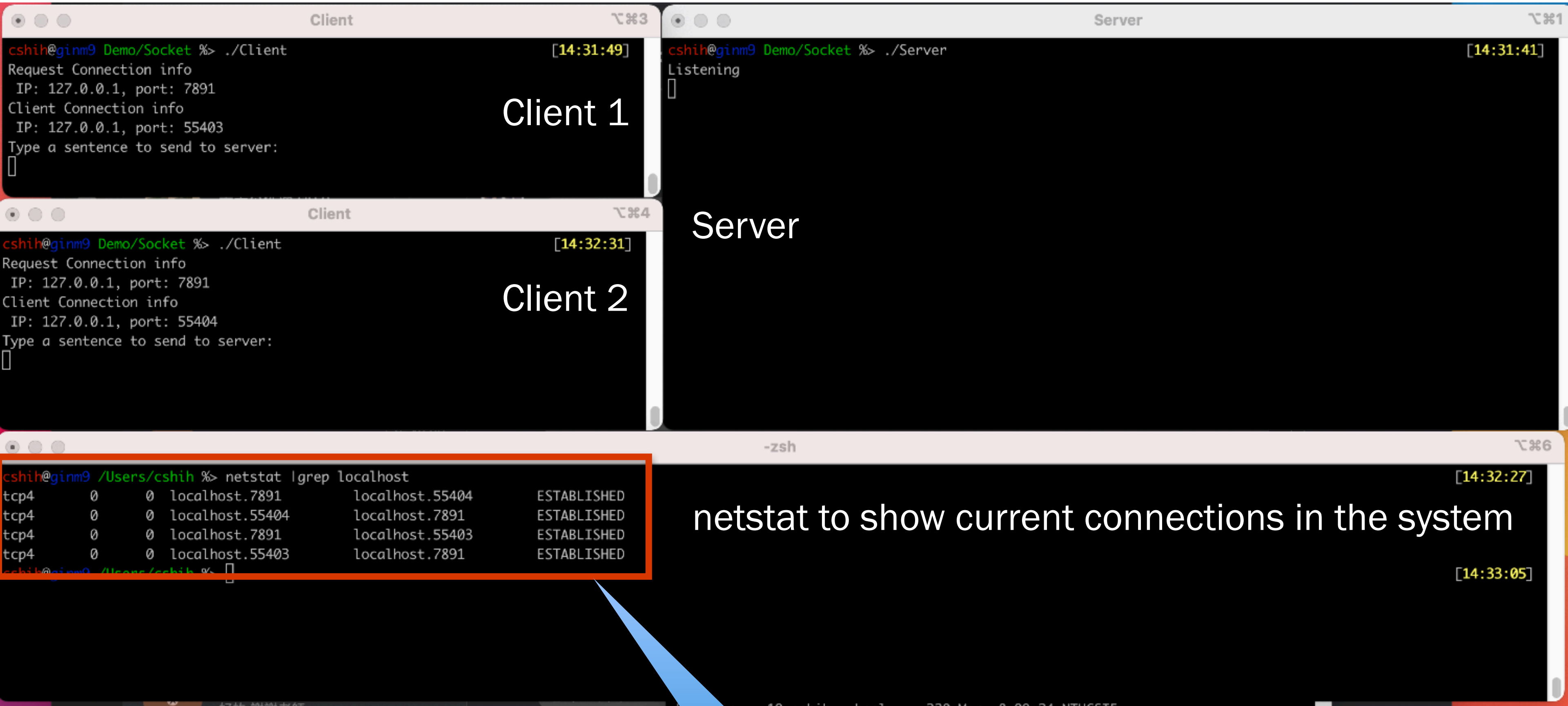
Socket Example: Client

```
int main(){
  int clientSocket, portNum, nBytes;
  char buffer[1024];
  struct sockaddr_in serverAddr, clientAddr;
  socklen_t addr_size;
  char ipaddr[20];

  clientSocket = socket(PF_INET, SOCK_STREAM, 0);

  portNum = 7891;

  memset(&serverAddr, '\0', sizeof(serverAddr));
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_port = htons(portNum);
  serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

  memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);
  addr_size = sizeof serverAddr;
  connect(clientSocket, (struct sockaddr *) &serverAddr, addr_size);
  printf("Request Connection info\n IP: %s, port: %d\n",
inet_ntoa(serverAddr.sin_addr), ntohs(serverAddr.sin_port));

  int len = sizeof(clientAddr);
  getsockname(clientSocket, (struct sockaddr *) &clientAddr, &len);
  printf("Client Connection info\n IP: %s, port: %d\n", inet_ntoa(clientAddr.sin_addr),
ntohs(clientAddr.sin_port));
```

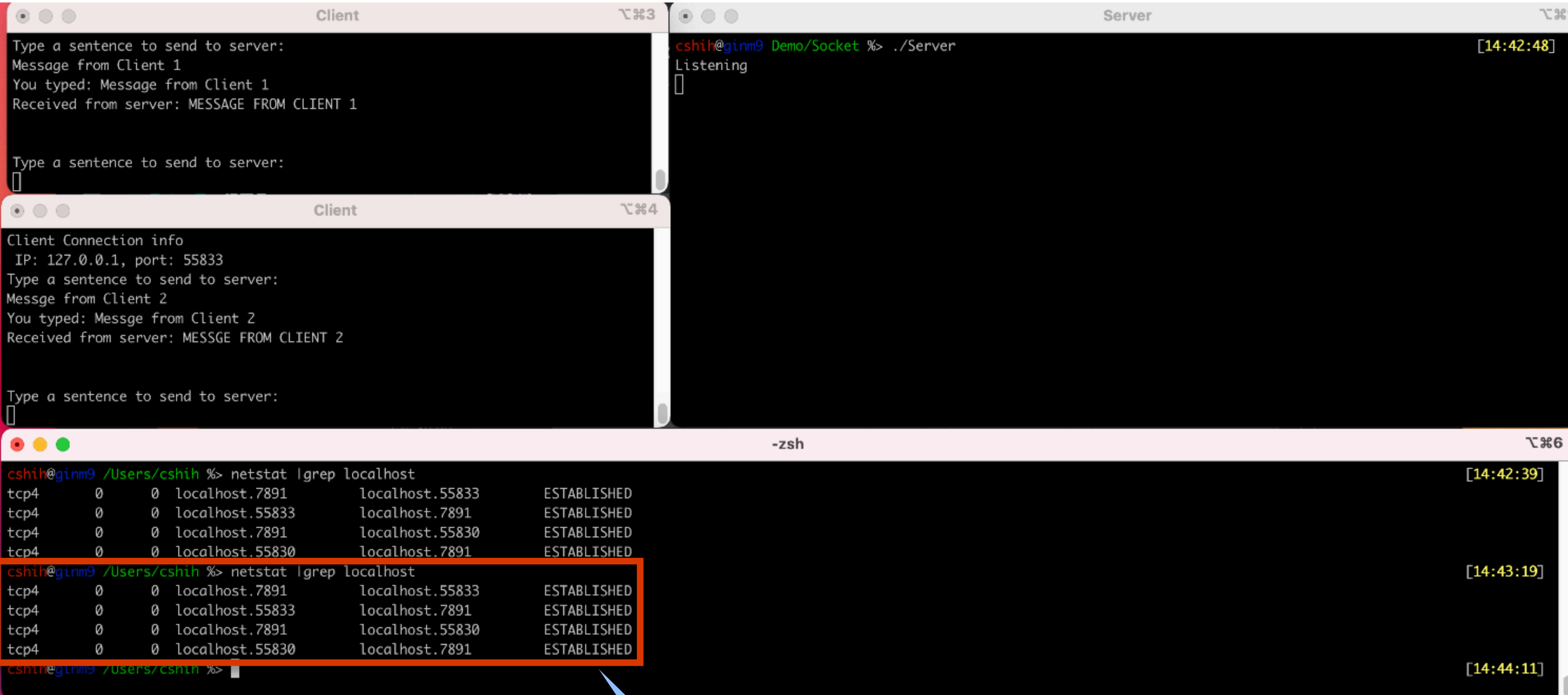
Socket Example: Client

```
  while(1){
    printf("Type a sentence to send to server:\n");
    fgets(buffer,1024,stdin);
    if (strlen(buffer)==1) break;
    printf("You typed: %s",buffer);
    nBytes = strlen(buffer) + 1;
    send(clientSocket,buffer,nBytes,0);
    recv(clientSocket, buffer, 1024, 0);
    printf("Received from server: %s\n\n",buffer);
  }

  return 0;
}
```

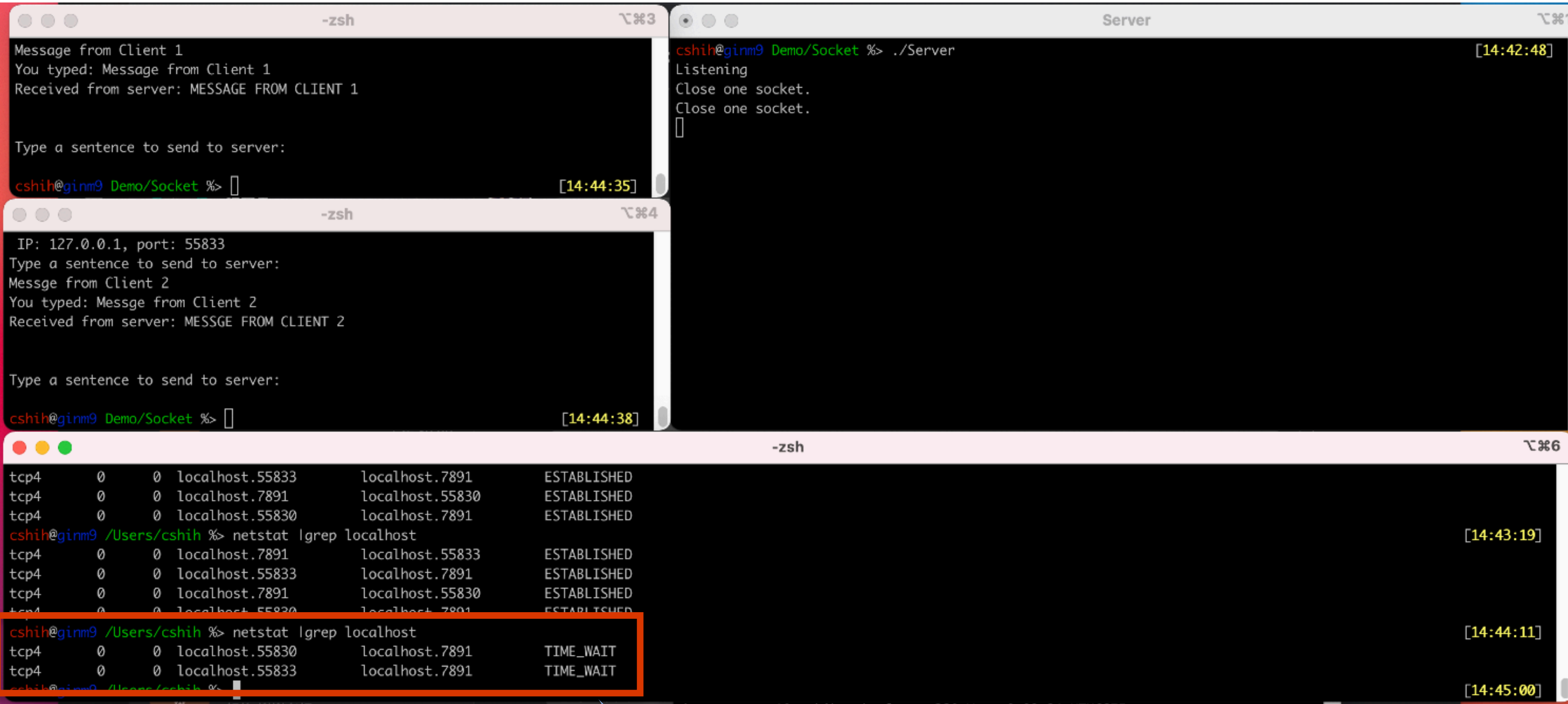
Results



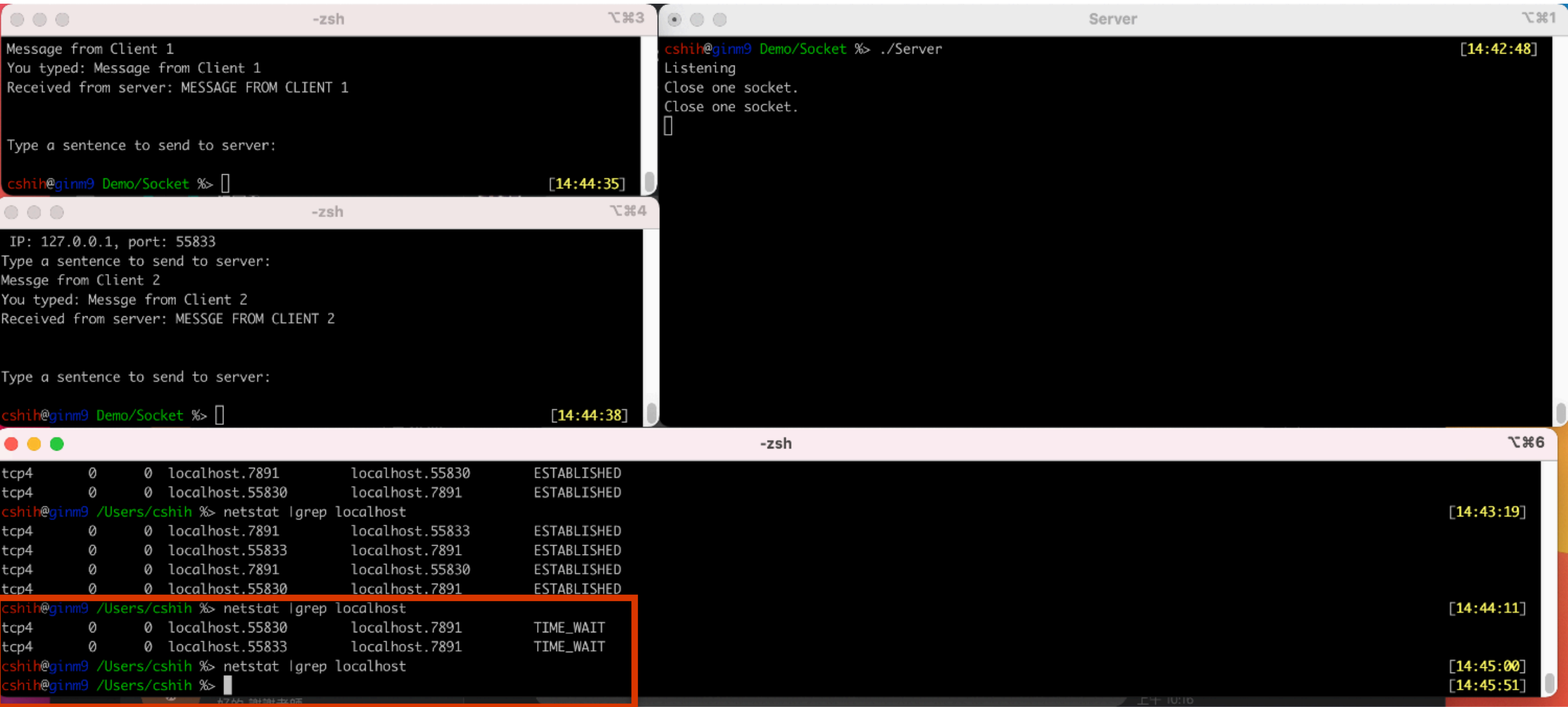
Two directional connections are established to each pair of client and server.



Two directional connections are established to send and receive messages.



Connections from servers (7891->x) are closed. The others are waiting to be released.

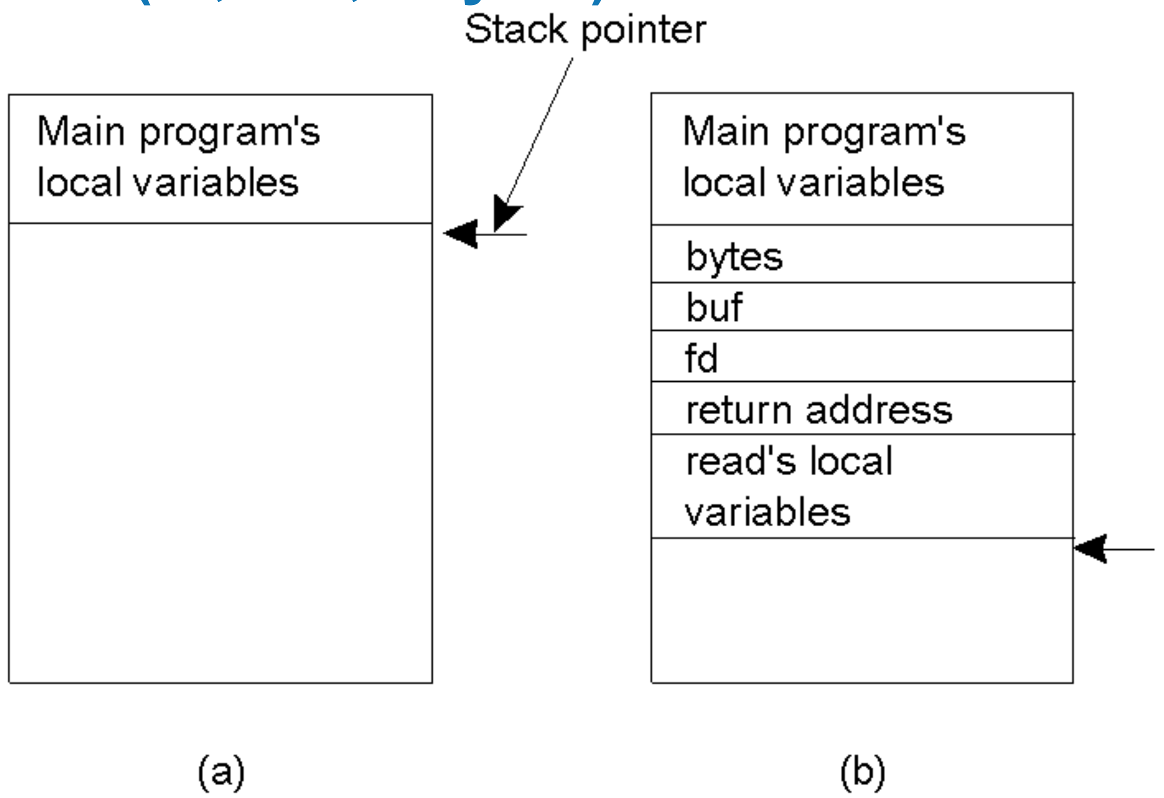


Connections are all released.

NEWS Lab 嵌入式系統暨無線網路實驗室

Conventional Procedure Call

count = read(fd, buf, nbytes)



a) Parameter passing in a local procedure call: the stack before the call to read

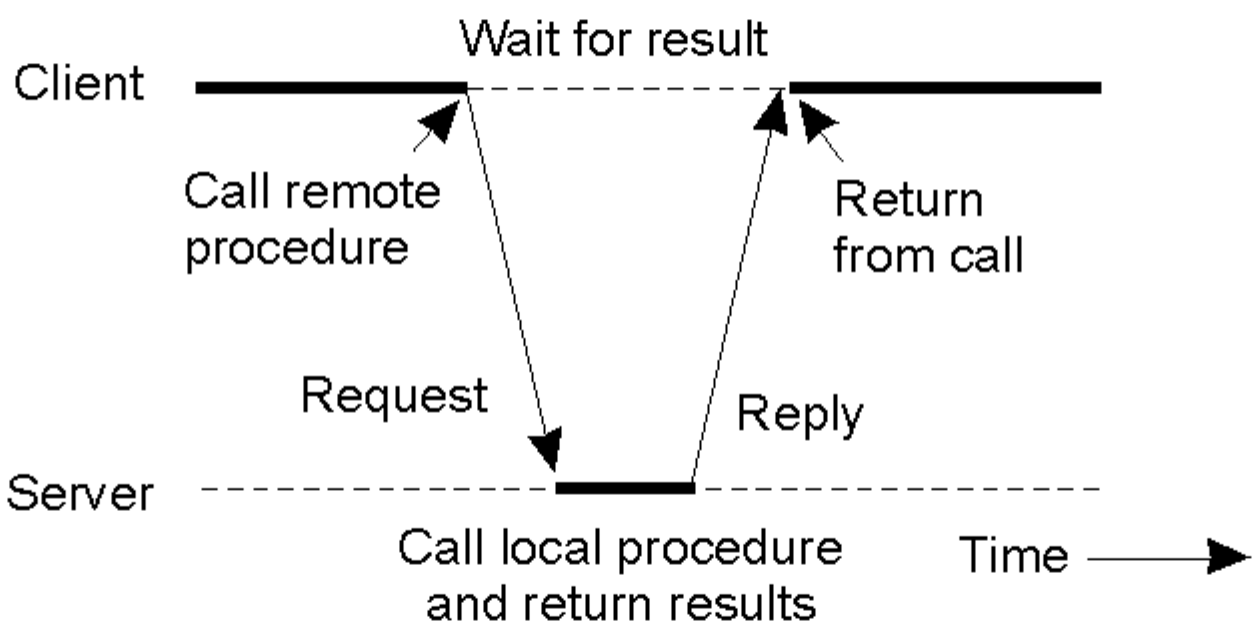
b) The stack while the called procedure is active

Remote Procedure Calls

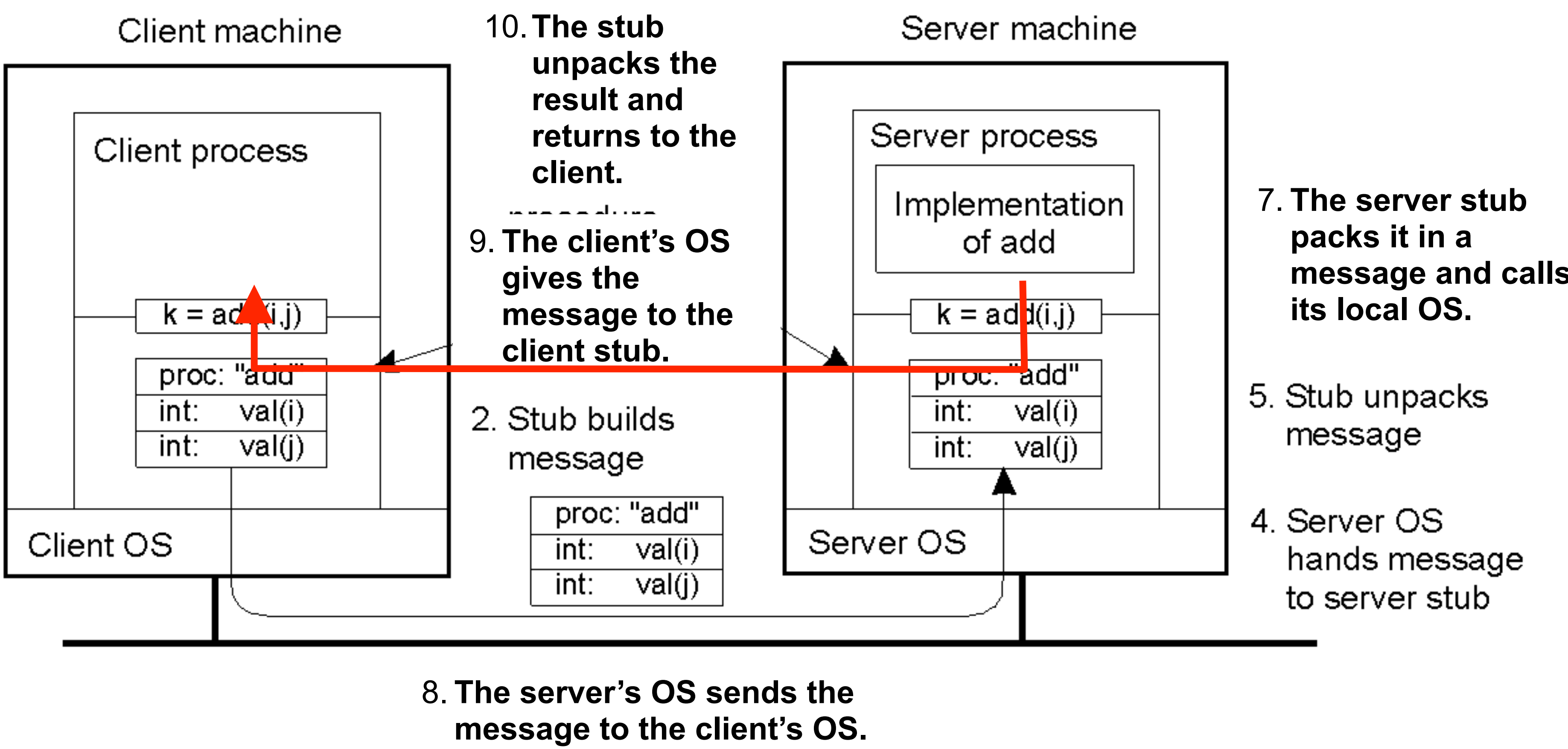
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- Stubs – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and marshalls the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in Microsoft Interface Definition Language (MIDL)

Remote Procedure Call

- The caller and called procedures are located on different processors/machines.
- No message passing is visible to the programmers.



Steps of a Remote Procedure Call



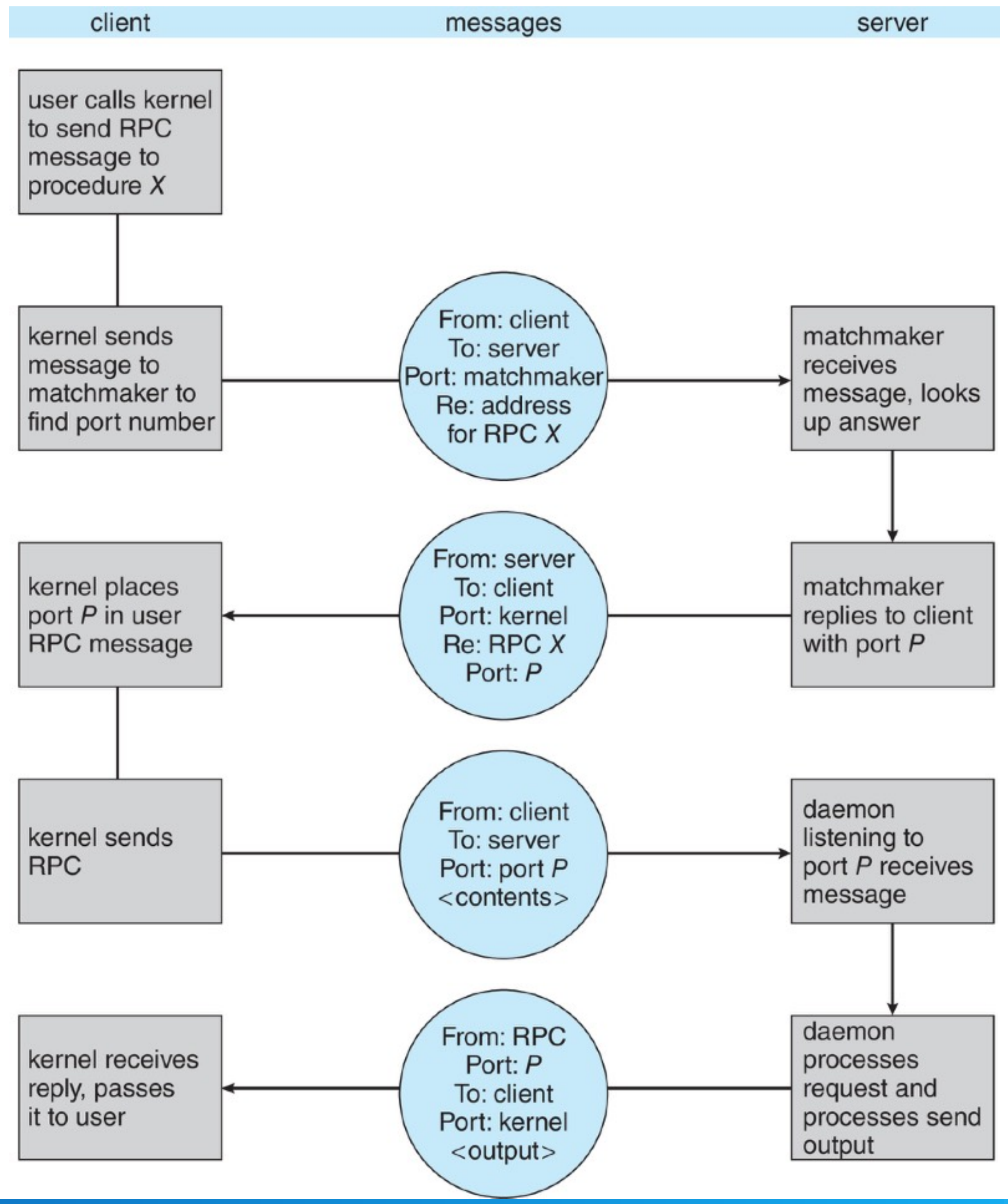
Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Remote Procedure Calls (Cont.)

- Data representation handled via External Data Representation (XDL) format to account for different architectures
  - Big-endian and little-endian
- Remote communication has more failure scenarios than local
  - Messages can be delivered exactly once rather than at most once
- OS typically provides a rendezvous (or matchmaker) service to connect client and server

Execution of RPC



Any Questions?

See You
Next Class

NEWS Lab
嵌入式系統暨無線網路實驗室

End of Chapter 3