Operating Systems

Chapter 2 Process and Threads

Tien Pham Van, Dr. rer. nat.

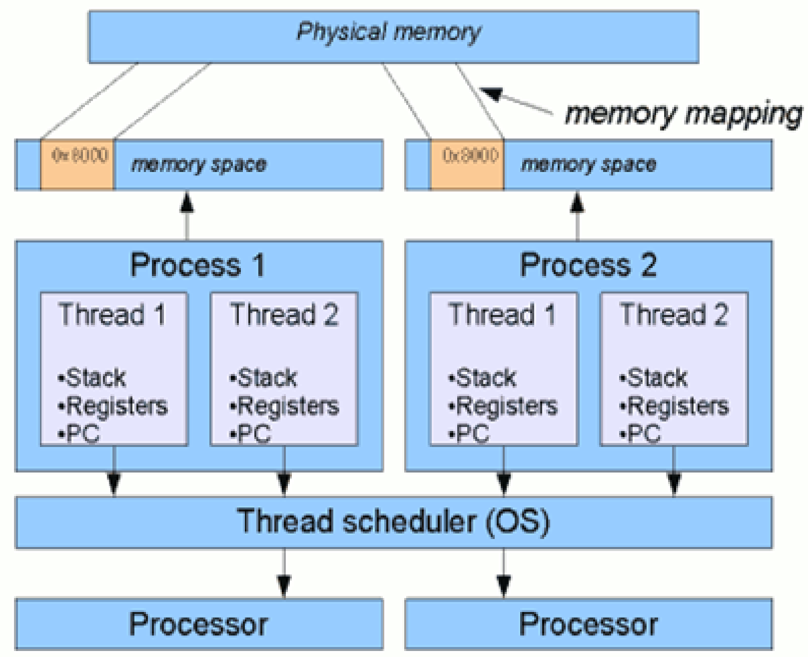
(*Lecture compiled with reference to other presentations*)

Process Concept

- A **process** is an instance of a computer program that is being executed. It contains the program code and its current activity.
- Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently

Processes

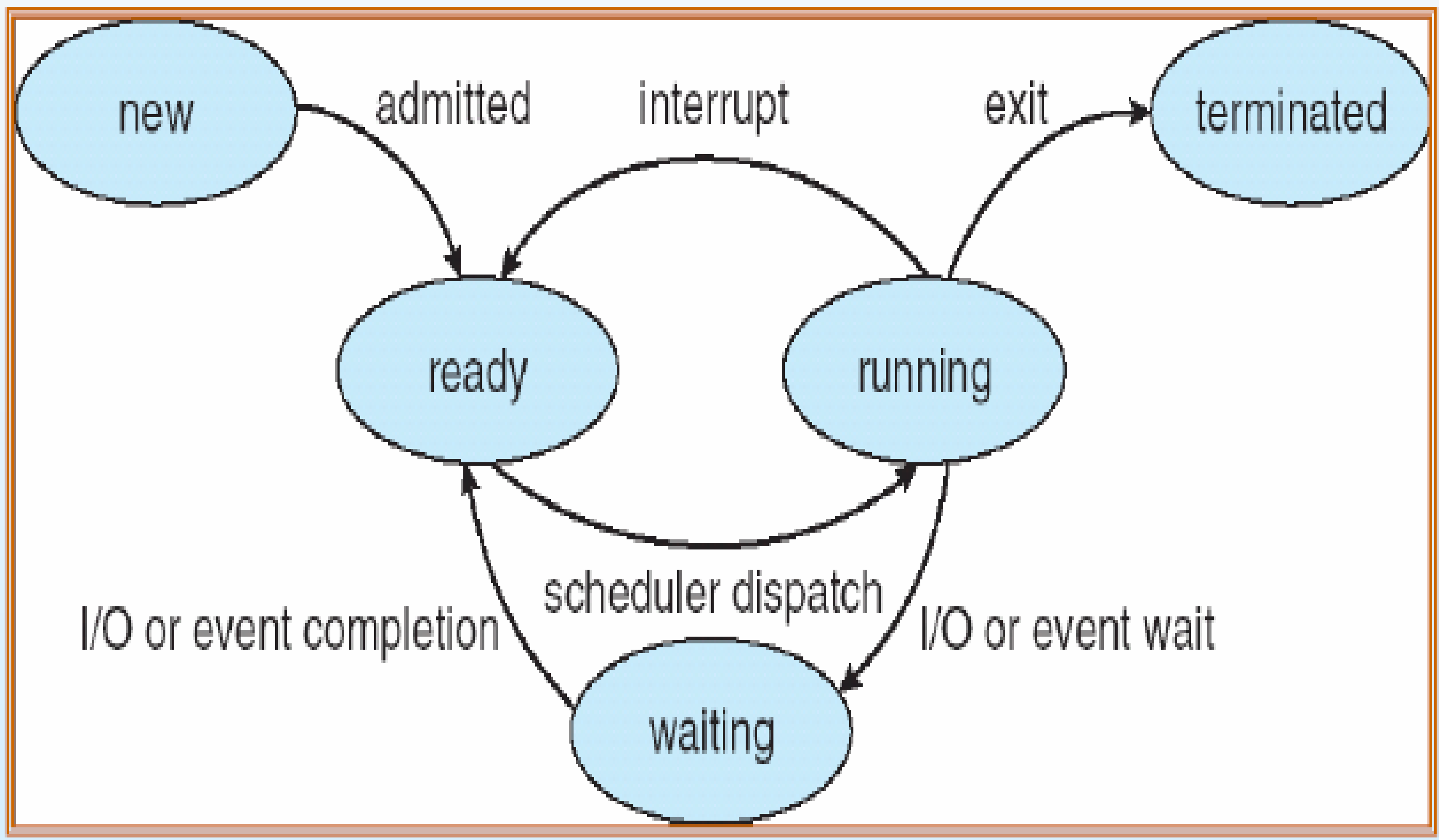
- A process is a unique execution of a program.
  - Several copies of a program may run simultaneously or at different times.
- A process has its own state:
  - registers;
  - memory.
- The operating system manages processes.



Process 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 process
- terminated: The process has finished execution

Diagram of Process State

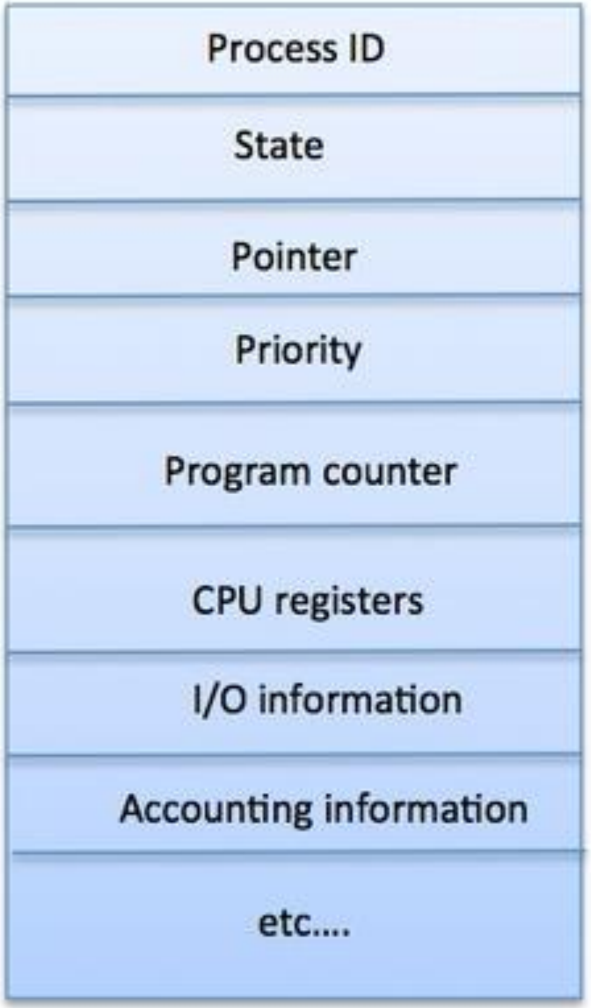


Process operations

https://www.youtube.com/watch?v=vLwMl9qK4T8

Process Control Block (PCB)

- Each process is represented in a operating system by a Process Control Block (PCB)
  - Process identifier
  - Process state
  - Program counter (PC)
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information



Process Control Block Process Identification

- Process Control Block

  The collection of attributes is refereed to as process control block.

- Unique numeric identifier
  - may be an index into the primary process table

Process Control

- Process Control Block (PCB)
- Process State Information
- Process Control Information
- Functions of an Operating-System Kernel
- Switch a Process
- Change of Process State
- Execution of the Operating System

Memory Tables

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

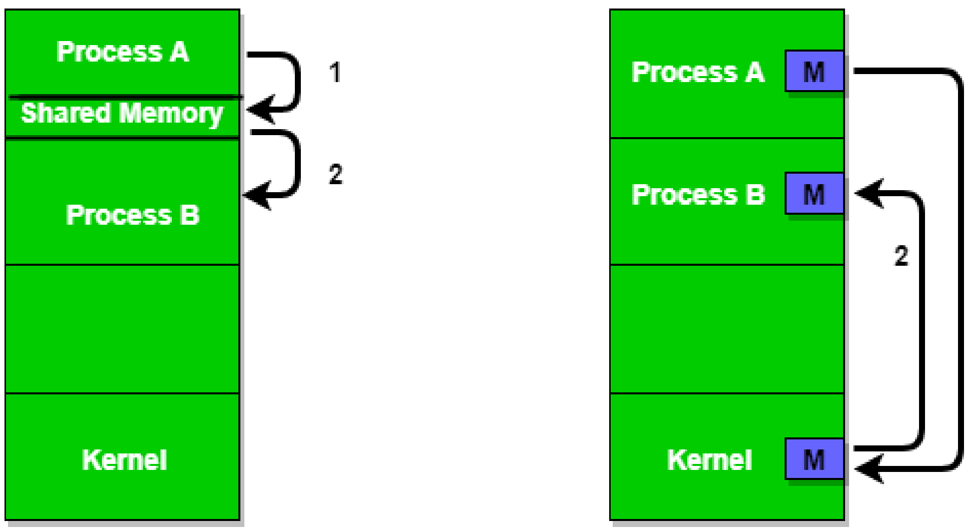
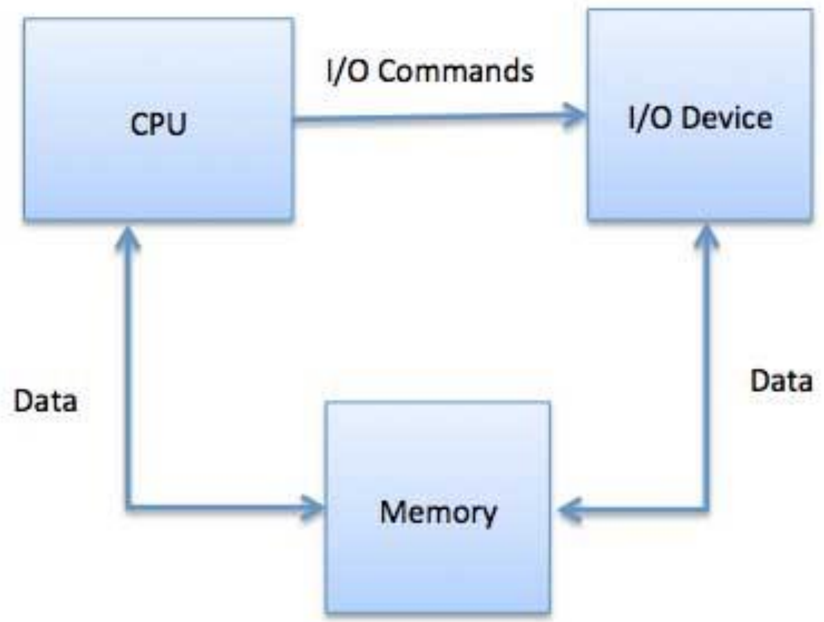


Figure 1 - Shared Memory and Message Passing

I/O Tables

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



File Tables

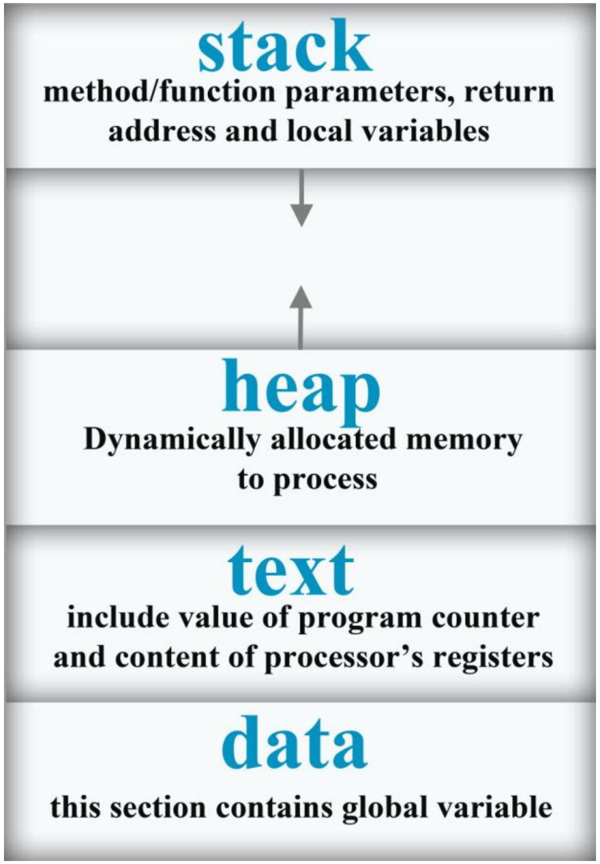
- Existence of files
- Location on secondary memory
- Current Status
- Attributes
- Sometimes this information is maintained by a file-management system

  (Think about *struct stat* and *stat( )*)

Process

Process: a program in execution

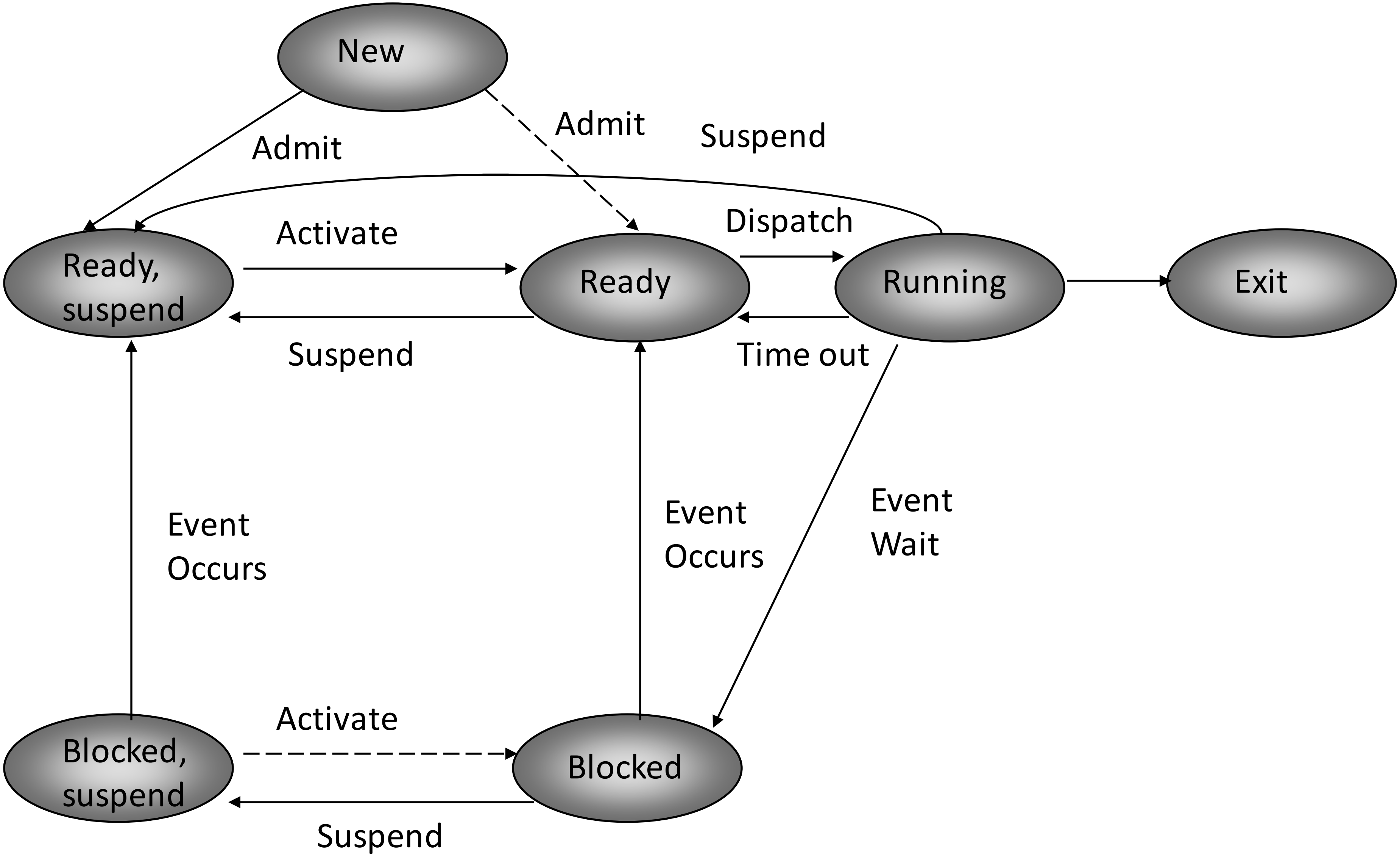
- Text section:
  - program code
  - program counter (PC)
  - data of registers
- Stack: to save temporary data
- Data section: store global variables
- Heap: for memory management



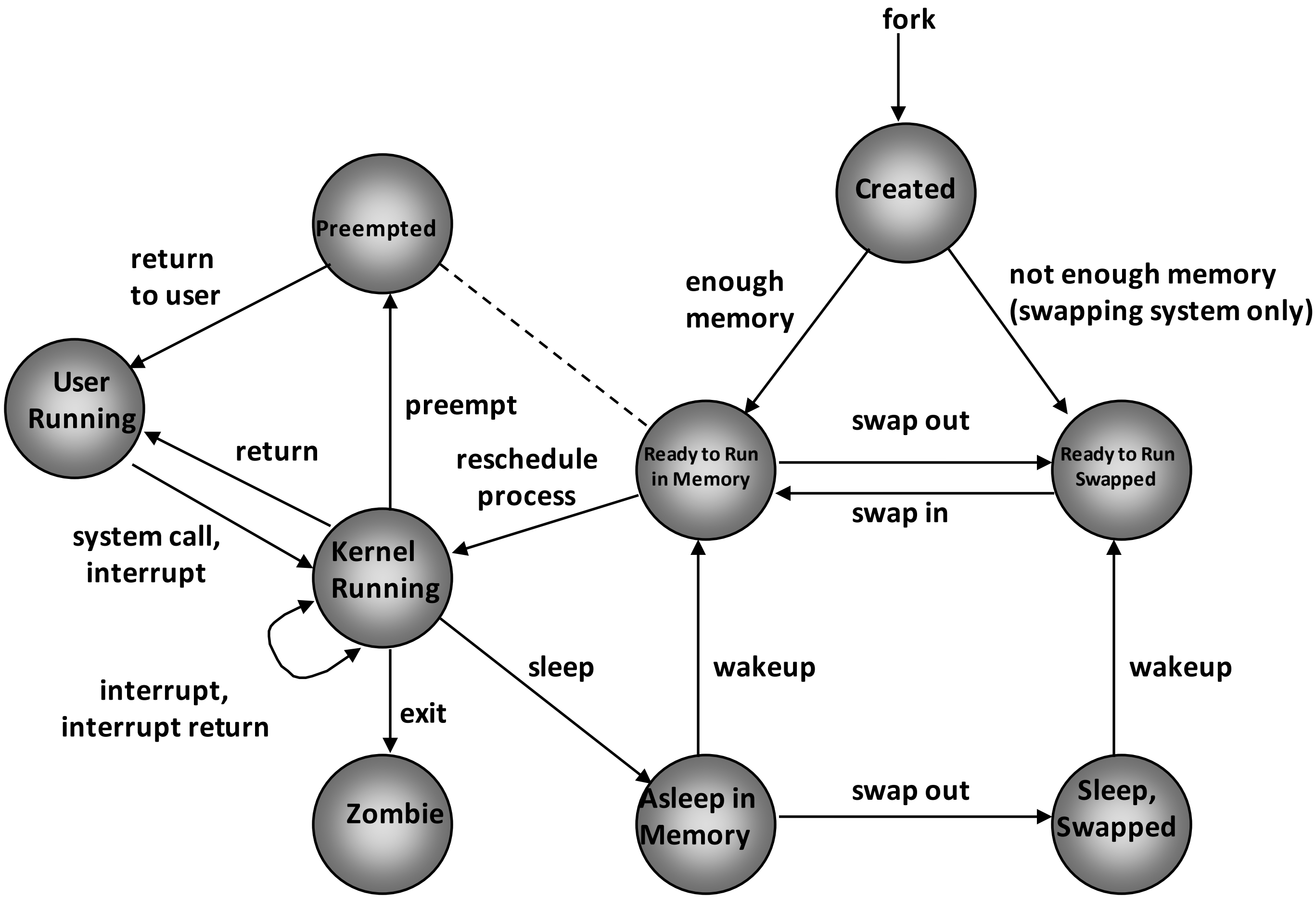
Further in programming...

- https://www.youtube.com/watch?v=n5lgcKch3Hk

Process State Transition Diagram with Two Suspend States - Seven-State Process Model



UNIX Process State Transition Diagram (1)



UNIX Process State Transition Diagram (2)

- User running: Executing in user mode.
- Kernel running: Executing in kernel model.
- Ready to run, in memory: Ready to run as soon as the kernel schedules it.

UNIX Process State Transition Diagram (3)

- Asleep in memory: unable to execute until an event occurs; process in main memory.
- Ready to run, swapped: process is ready to run, but the the swapper must swap the process into main memory before the kernel can schedule it to execute.
- Sleeping, swapped: The process is awaiting an event and has been swapped to secondary storage.

- Preempted: process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
- Created: process is newly created and not yet ready to run.
- Zombie: process no longer exists, but it leaves a record for its parent process to collect.

UNIX Process Control Table

- Process Identifiers

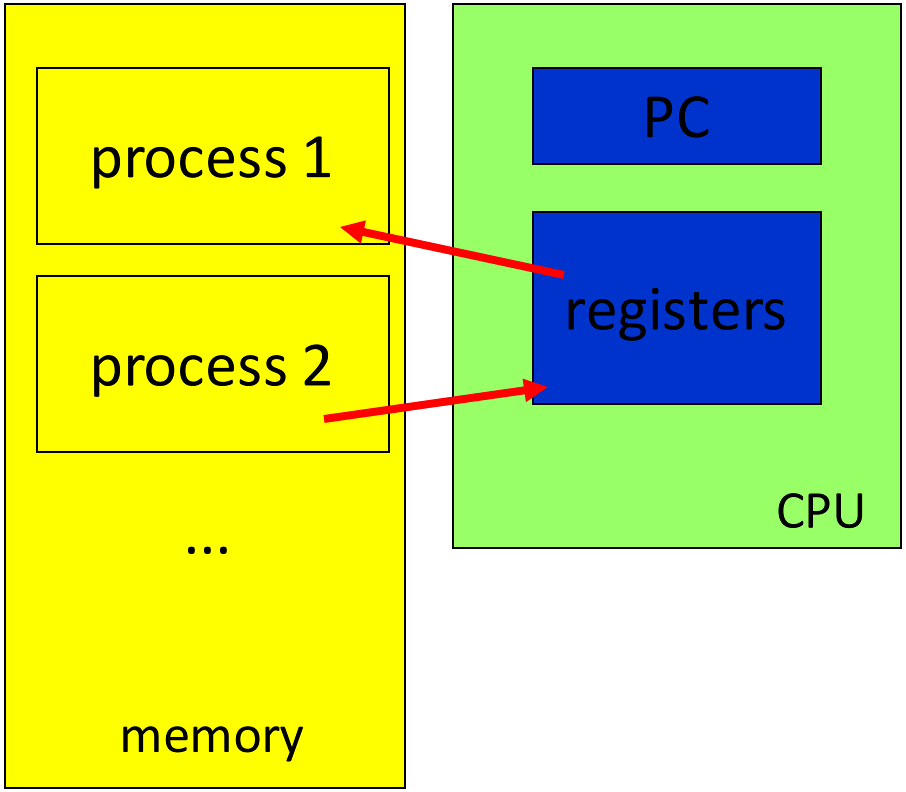
  ID of this process and ID of parent process.
- User Identifiers

  real user ID, effective user ID
- Pointers

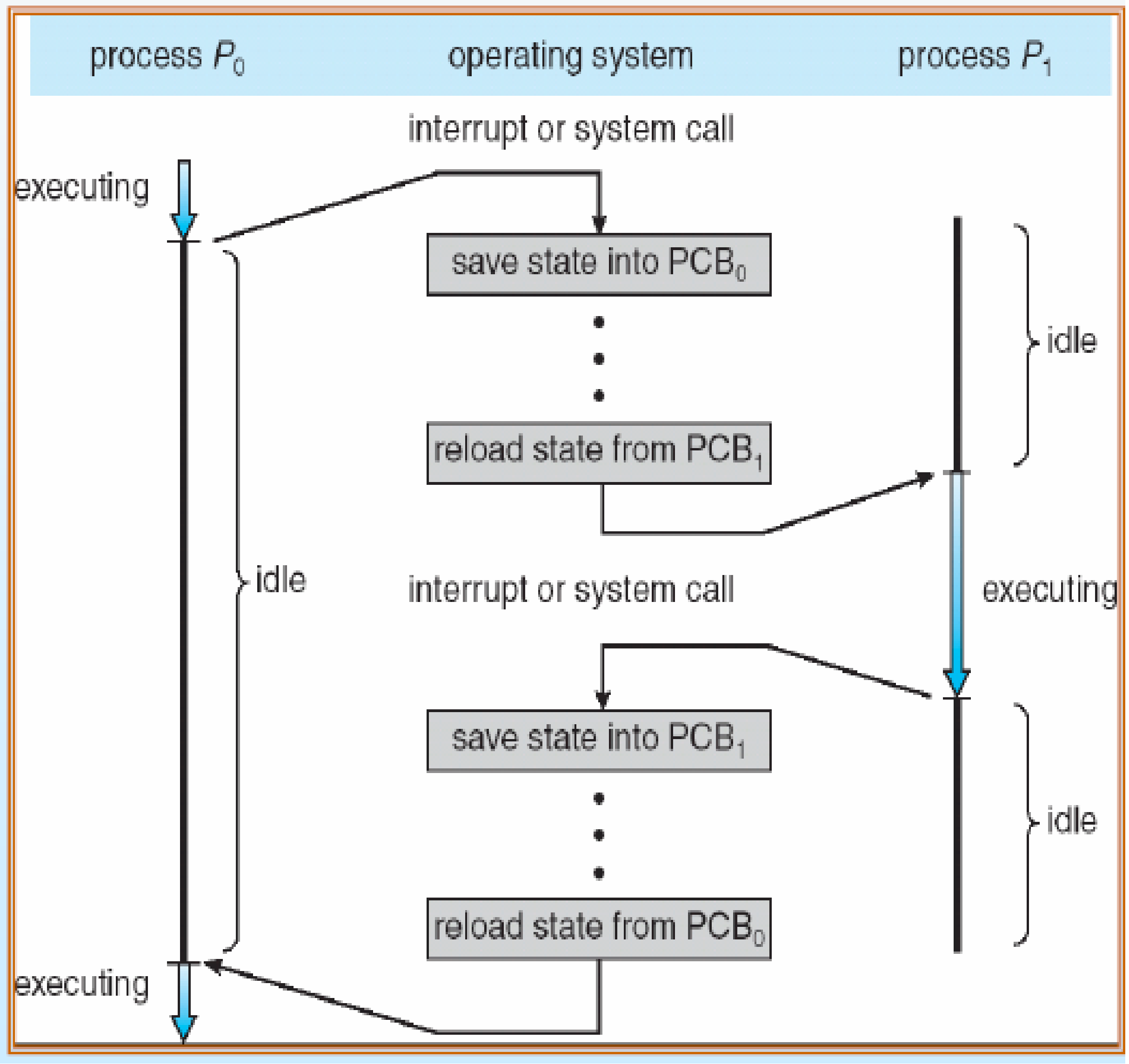
  To user area and process memory (text, data, stack)
- Process Size, Priority, Signal, Timers, ...

Context switch

- Activation record: copy of process state.
- Context switch:
  - current CPU context goes out;
  - new CPU context goes in.



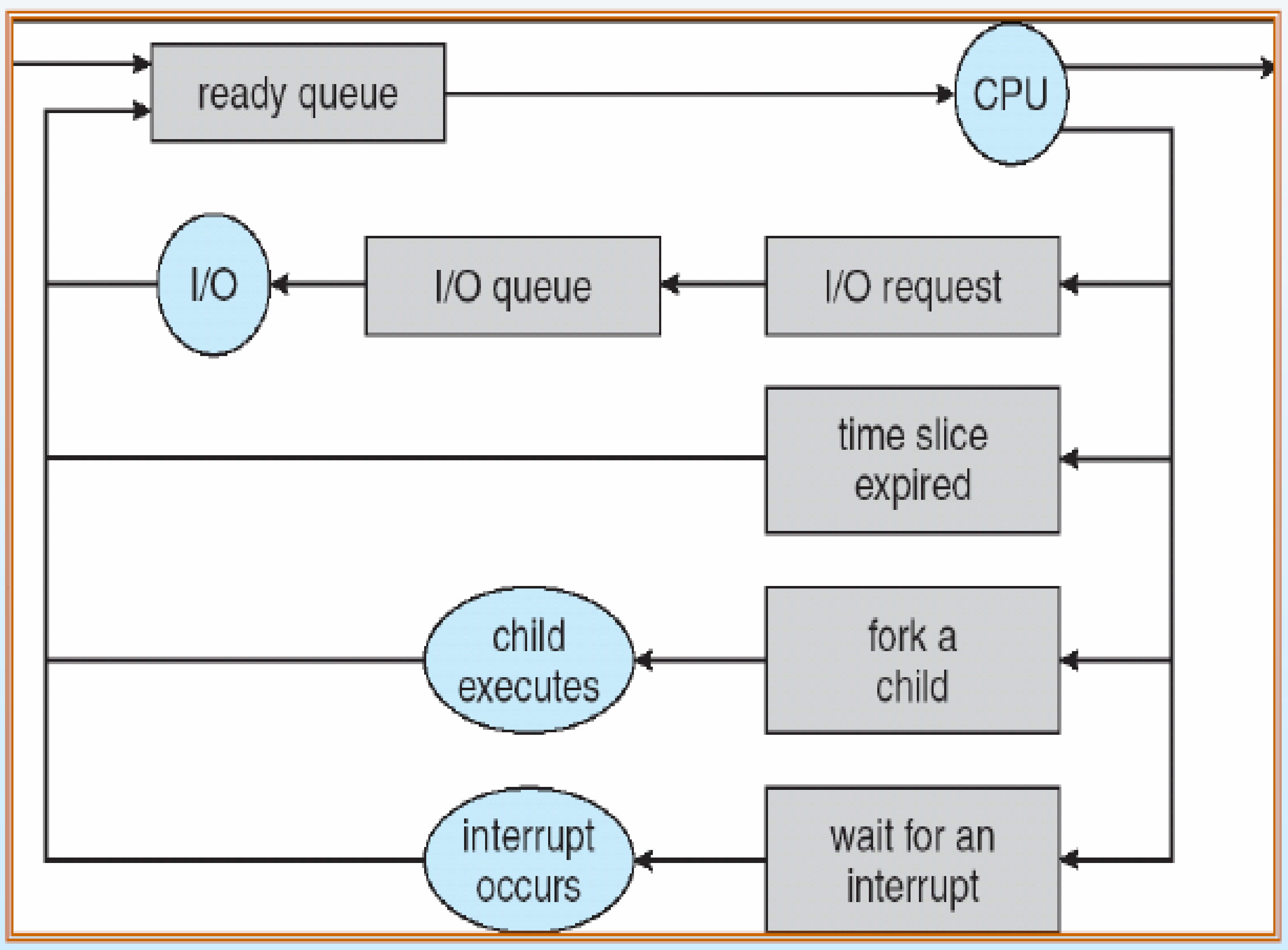
CPU Switch From Process to Process



Process Scheduling

- As processes entered the system, they are put into job queue
- The processes that stay in main memory and are ready and waiting to execute are kept on a list called ready queue
- A ready queue contains pointers to the first and final PCBs in the list
- The list of processes waiting for a particular I/O is called a device queue

Queueing Diagram



Schedulers

- Long-term scheduler

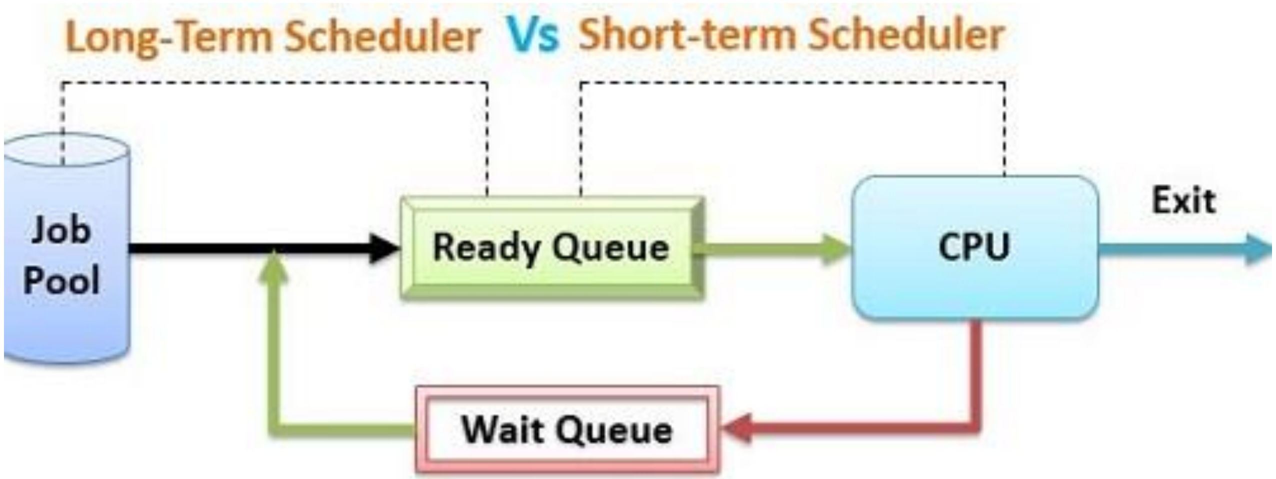
  (or job scheduler) –selects which processes should be brought into the ready queue

- Short-term scheduler

  (or CPU scheduler) –selects which process should be executed next and allocates CPU

Schedulers

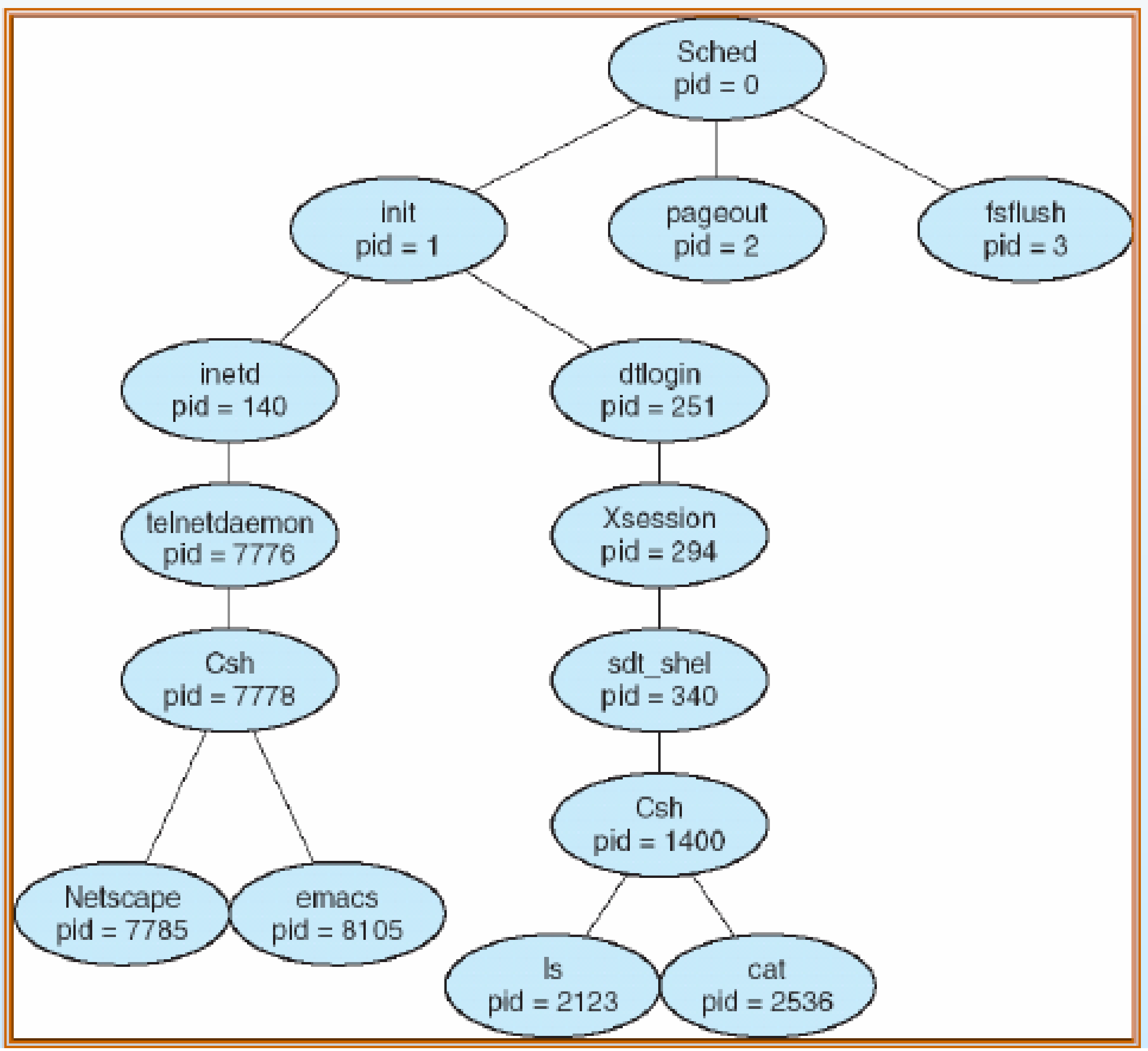
- Short-term scheduler is invoked very frequently (milliseconds) ⇒(must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒(may be slow)
- The long-term scheduler controls the *degree of multiprogramming*



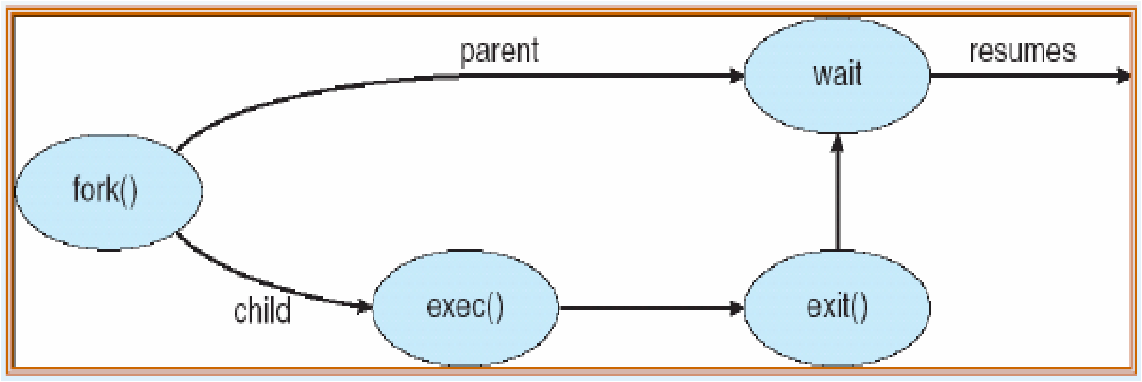
Process Creation

- A process may create several new processes. The creating process is called a parent process, and new processes are called children process
- Each of these processes may create other processes, forming a tree processes

Process Tree



Process Creation



C Program Forking Separate Process

```
int main()
{
Pid_t  pid;
        /* fork another process */
        pid = fork();
        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 */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }
}
```

Co-operative multitasking

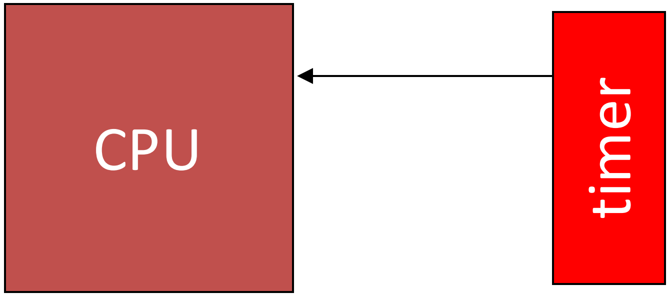
- Improvement on co-routines:
  - hides context switching mechanism;
  - still relies on processes to give up CPU.
- Each process allows a context switch at cswitch() call.
- Separate scheduler chooses which process runs next.

Problems with co-operative multitasking

- Programming errors can keep other processes out:
  - process never gives up CPU;
  - process waits too long to switch, missing input.

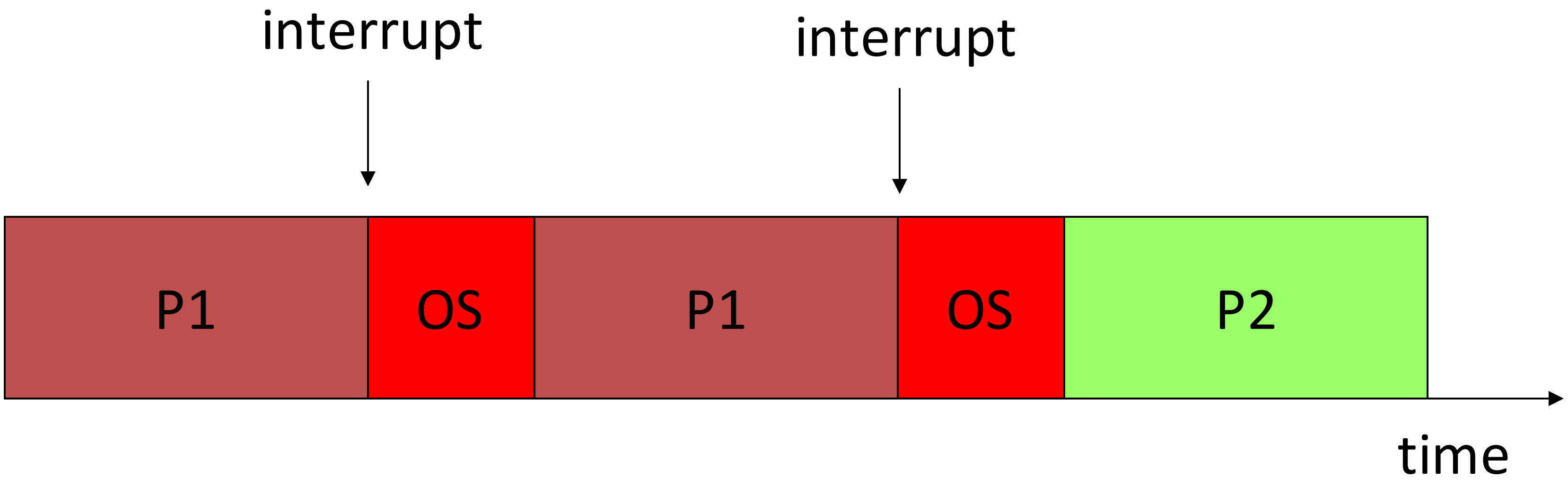
Preemptive multitasking

- Most powerful form of multitasking:
  - OS controls when contexts switches;
  - OS determines what process runs next.
- Use timer to call OS, switch contexts:



Preemptive context switching

- Timer interrupt gives control to OS, which saves interrupted process's state in an activation record.
- OS chooses next process to run.



When to Switch a Process

- Trap
  - error occurred
  - may cause process to be moved to Exit state
- Supervisor call
  - such as file open

When to Switch a Process

- Memory fault
  - memory address is in virtual memory so it must be brought into main memory
- Interrupts
  - Clock
    - process has executed for the maximum allowable time slice
  - I/O

Process Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

        /*  produce an item and put in
   nextProduced  */
        while (count == BUFFER_SIZE)
            ; // do nothing
          buffer [in] = nextProduced;
          in = (in + 1) % BUFFER_SIZE;
          count++;
}
```

Consumer

```
while (true)  {
        while (count == 0) ; // do nothing
          nextConsumed =  buffer[out];
          out = (out + 1) % BUFFER_SIZE;
                count--;

        /*  consume the item in nextConsumed */
 }
```

Race Condition

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

  https://www.youtube.com/watch?v=ZQb3DRy0g8U&ab_channel=Xoviabcs

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
            critical section
    flag[i] = FALSE;
            remainder section
} while (TRUE);
```

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
          critical section
    release lock
          remainder section
} while (TRUE);
```

TestAndndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
      {
           boolean rv = *target;
           *target = TRUE;
           return rv;
      }
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {
    while ( TestAndSet (&lock ))
        ;   // do nothing

        //    critical section

    lock = FALSE;

        //     remainder section

} while (TRUE);
```

compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected,
                     int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter "value"
3. Set the variable "value" the value of the passed parameter "new_value" but only if "value" =="expected". That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
- Solution:

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
          ; /* do nothing */
       /* critical section */
    lock = 0;
       /* remainder section */
 } while (true);
```

Lock implementation

1. [Stanford CS140]
2. https://www.scs.stanford.edu/23wi-cs212/notes/synchronization1.pdf

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    ```
    wait (S) {
        while S <= 0
            ; // no-op
          S--;
    }
    ```
  - signal (S) {
    ```
    signal (S) {
      S++;
    }
    ```

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
       // Critical Section
    signal (mutex);
       // remainder section
} while (TRUE);
```

- condition x;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is
    suspended.
  - x.signal () – resumes one of processes (if any) that
    invoked x.wait ()

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

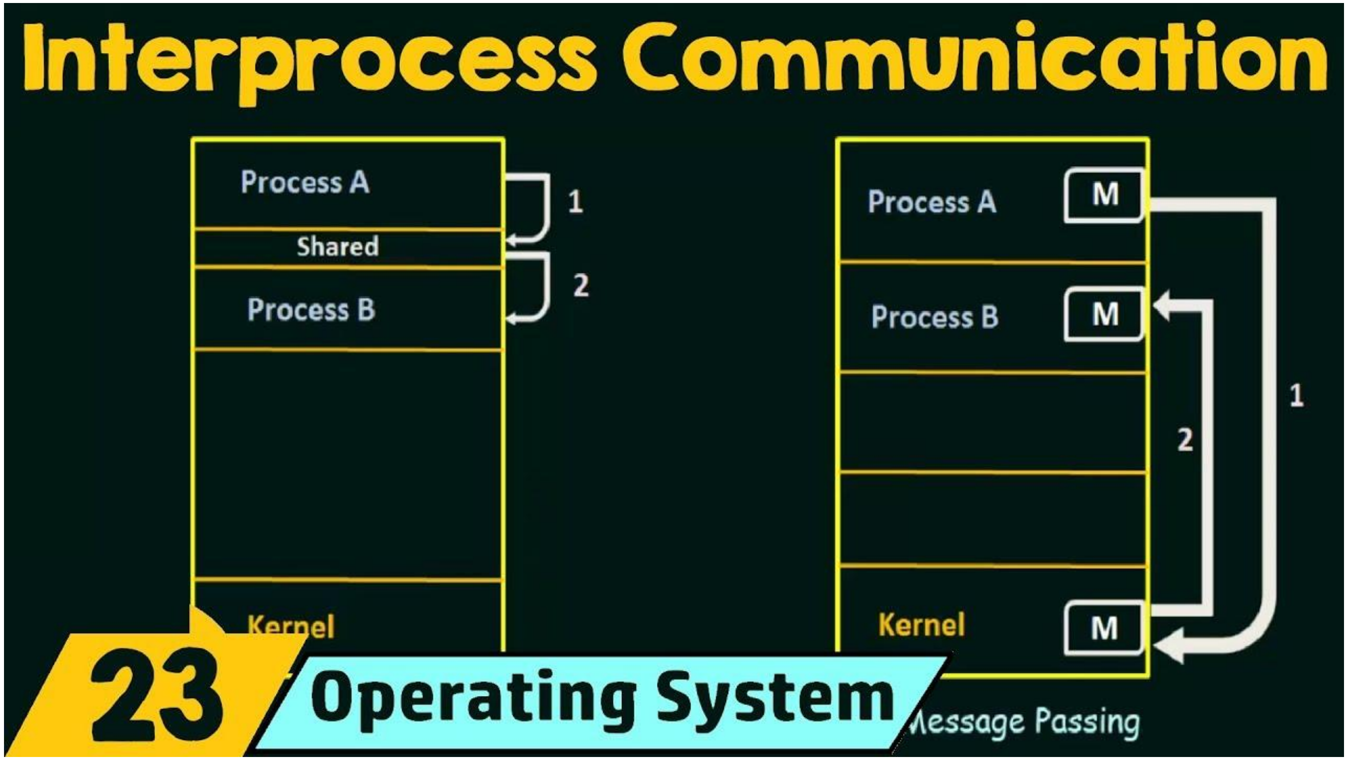
Deadlock and Starvation

https://www.scs.stanford.edu/23wi-cs212/notes/synchronization2.pdf

1. Open lecture: https://www.youtube.com/watch?v=exlaEOVRWQM
2. Stanford CS212: https://www.scs.stanford.edu/23wi-cs212/notes/scheduling.pdf

Inter-Process Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.



- Online courseware: https://www.youtube.com/watch?v=G2vwkBZy894

Synchronization in message passing (1)

- Message passing may be blocking or non-blocking.
- Blocking is considered synchronous
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- Non-blocking is considered asynchronous
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Synchronization in message passing (2)

- For the sender: it is more natural not to be blocked after issuing send:
  - can send several messages to multiple destinations.
  - but sender usually expect acknowledgment of message receipt (in case receiver fails).
- For the receiver: it is more natural to be blocked after issuing receive:
  - the receiver usually needs the information before proceeding.
  - but could be blocked indefinitely if sender process fails before send.

Shared Memory – POSIX

- POSIX Shared Memory example
- Process first creates shared memory segment

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

- Process wanting access to that shared memory must attach to it

```
shared_memory = (char *) shmat(segment_id, NULL, 0);
```

- Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

- When done a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

- Now process can remove the shared memory segment

```
shmdt(shared_id, IPC_RMID, NULL);
```

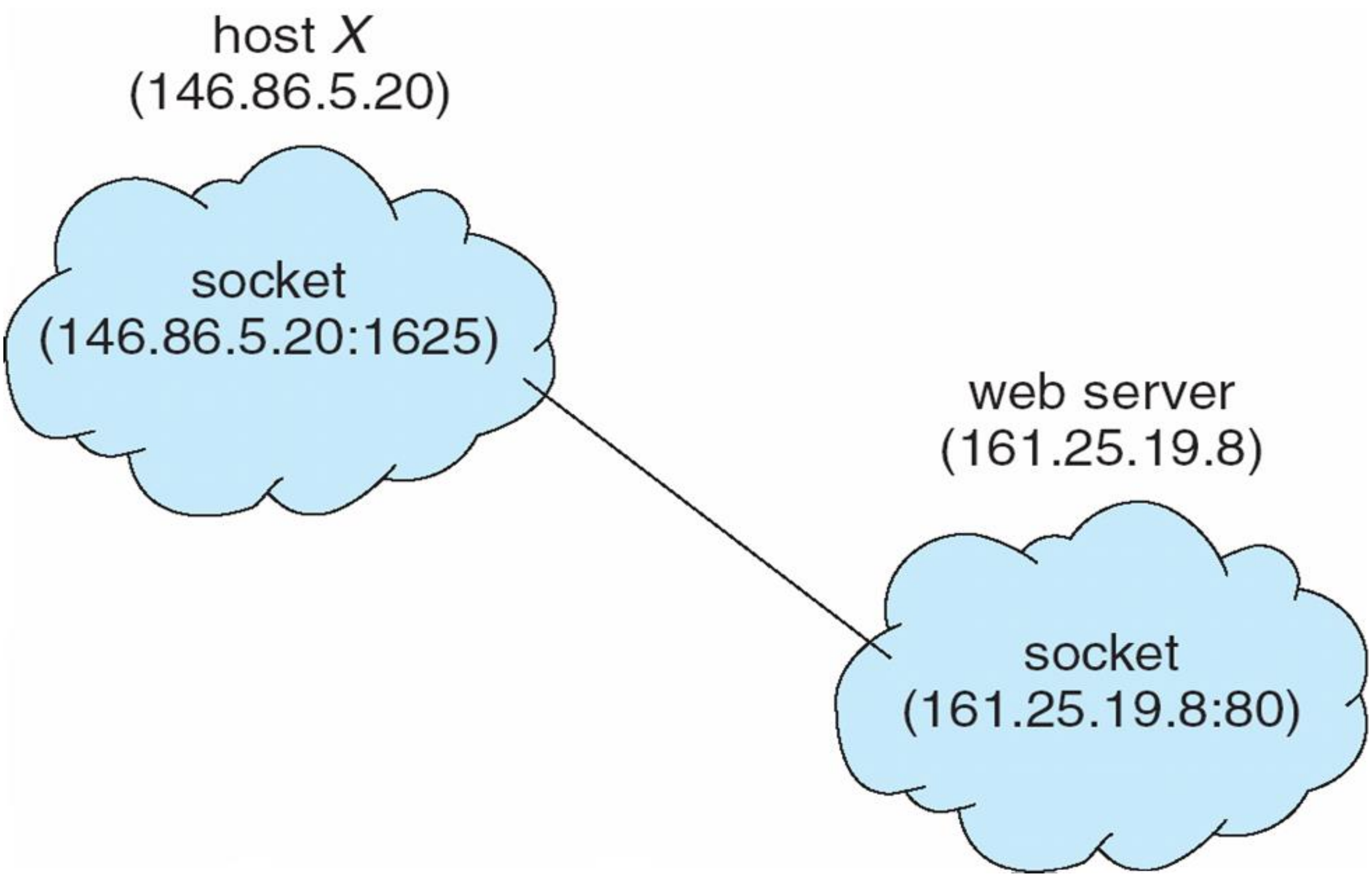
Communications in Client-Server Systems

- There are various mechanisms:

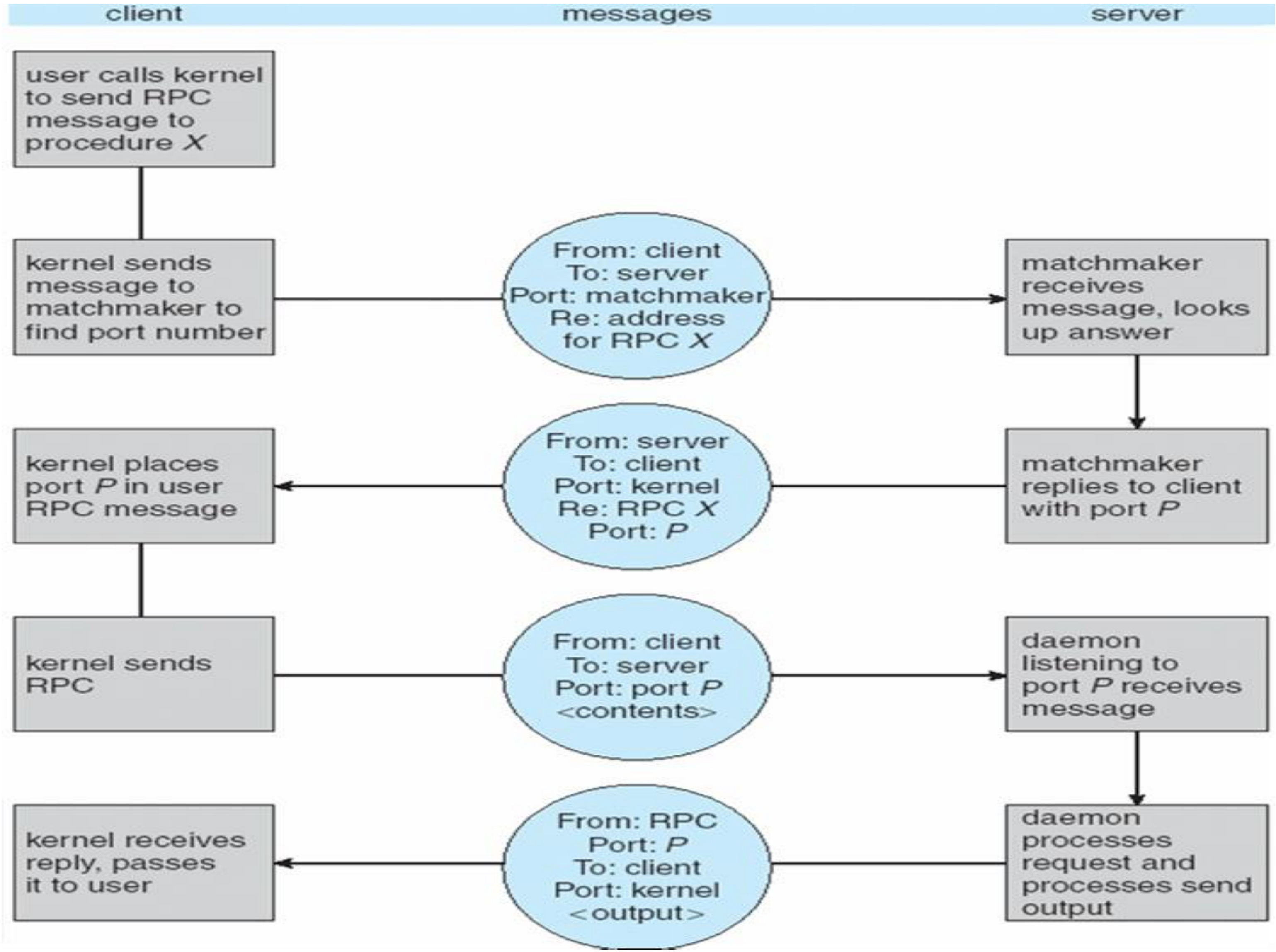
1. Sockets (Internet)
2. Remote Procedure Calls (RPCs)
3. Remote Method Invocation (RMI, Java)

Email: tien.phamvan1@hust.edu.vn
C9-411, Dai Co Viet str. 1, HBT, Hanoi
Tel: +84-243-8693596

Socket Communication



Remote Procedure Call ()RPC



Multithreading

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Benefits of multithreading

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
  - *S* is serial portion
  - *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example (Cont.)

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP (light-weight process – running on top of kernel thread to allow user space multitasking)
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Email: tien.phamvan1@hust.edu.vn
C9-411, Dai Co Viet str. 1, HBT, Hanoi
Tel: +84-243-8693596