Operating System Concepts

Che-Wei Chang
chewei@mail.cgu.edu.tw
Department of Computer Science and Information Engineering, Chang Gung University

Contents

Chapter 3. Process Concept

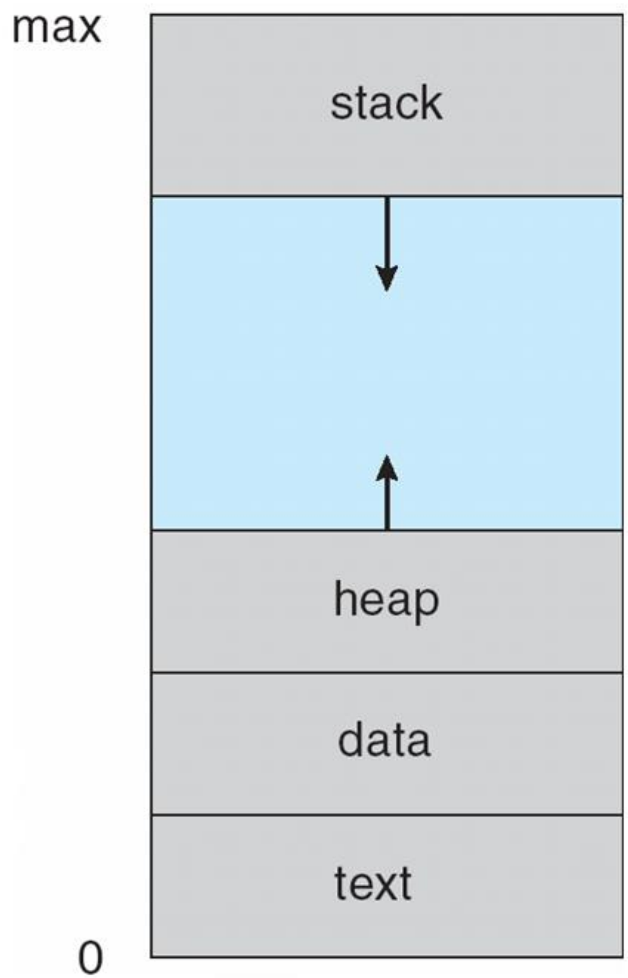
Objectives

- To introduce the notion of a process
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore inter-process communication
- To describe communication in client-server systems

Basic Process Concept

- A program is a **passive** entity stored on disk, and a process is an **active** entity
  - A program becomes process when the executable file is loaded into memory
  - The execution of a program started via GUI mouse clicks, the command line entry of its name, etc.
  - One program can be executed as several processes
- An operating system can execute a variety of programs
  - In batch systems: jobs
  - In time-shared systems: user programs or tasks

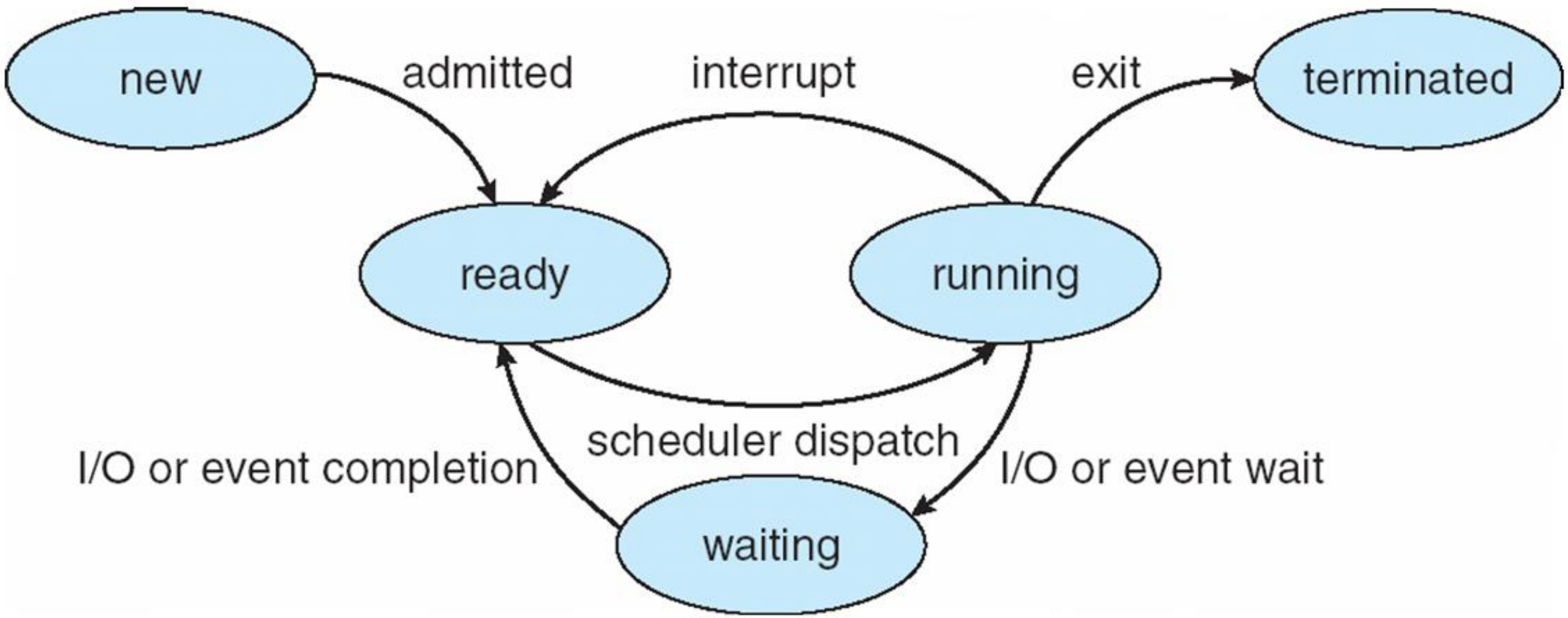
Process in Memory



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 processor
- **Terminated**: The process has finished execution

Diagram of Process States

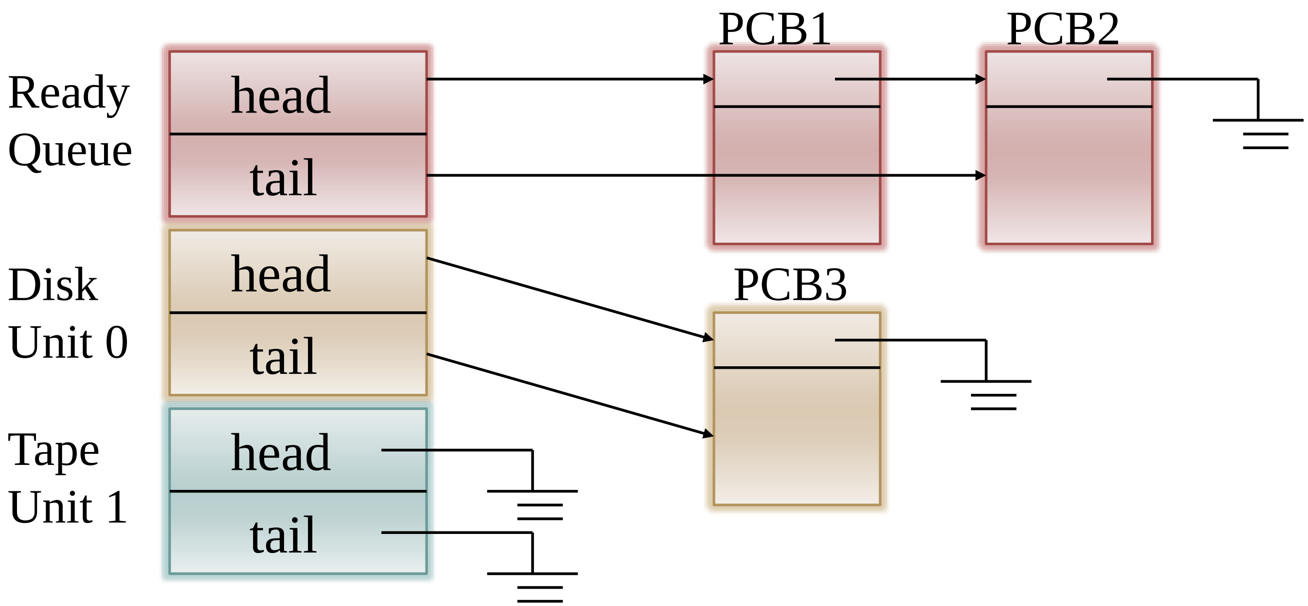


Process Control Block (PCB)

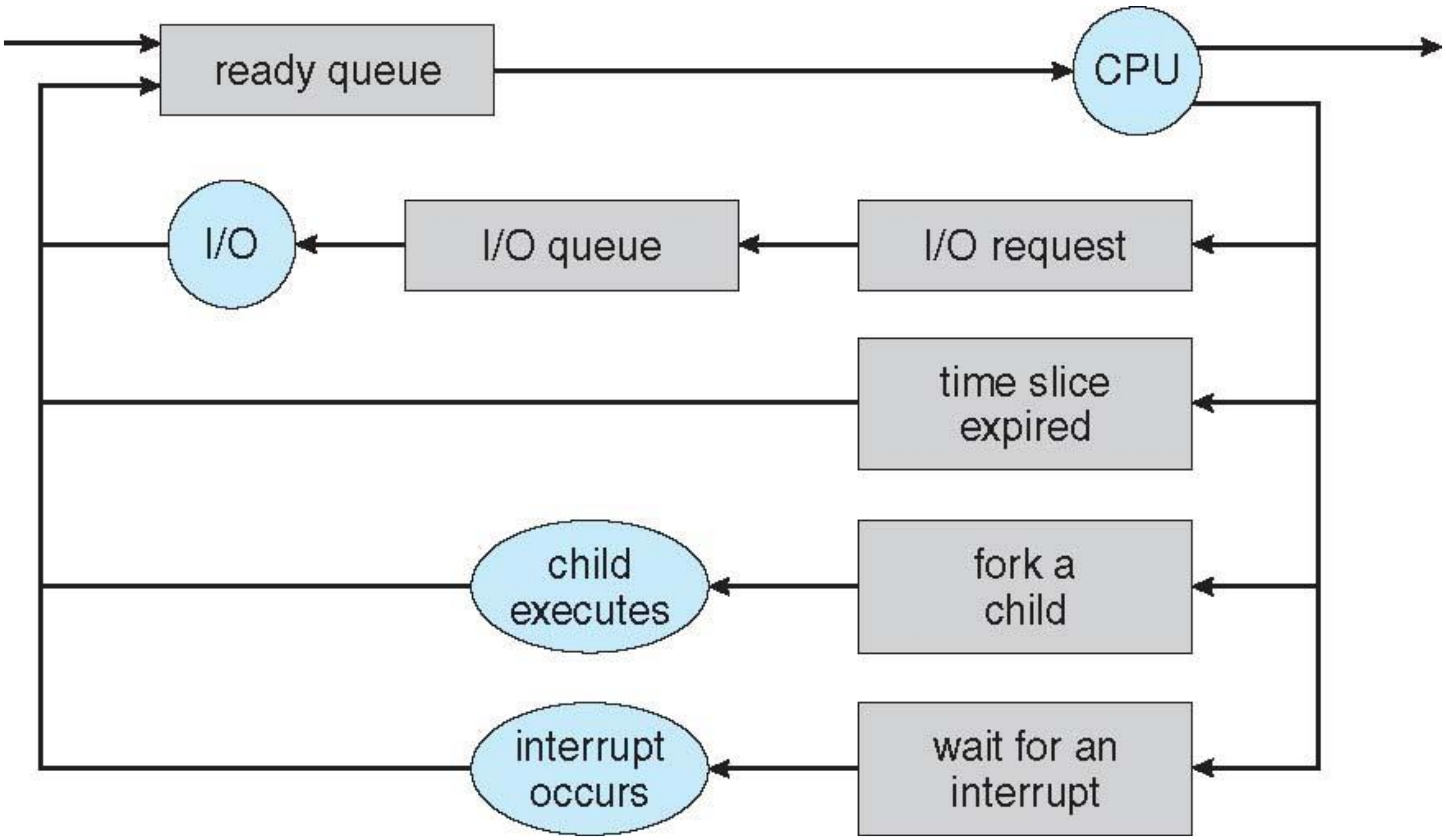
- PCB: The repository for any information that may vary from process to process
  - Process state– running, waiting, etc
  - Program counter– location of the currently executed instruction
  - CPU registers– contents of all process-centric registers
  - CPU scheduling information– priorities, scheduling 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 opened files

Process Scheduling with PCB

- The goal of multiprogramming
  - Maximize CPU/resource utilization
- The goal of time sharing
  - Allow each user to interact with his/her program



Process Scheduling- A Queueing Diagram

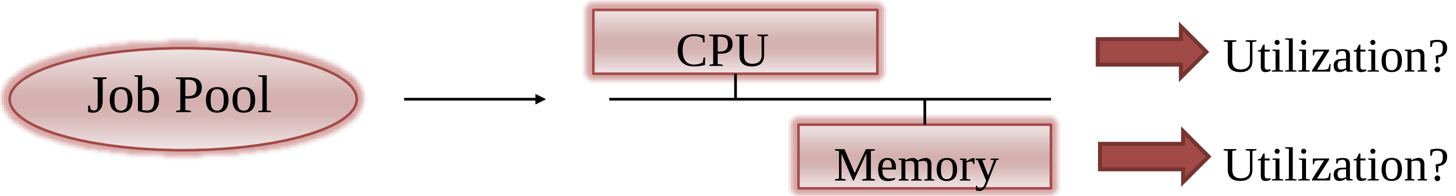


Processor Schedulers

- **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
- **Medium-term scheduler** can be added as **swapper**

Long-Term Scheduler

- Processes can be described as either:
  - **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
- Long-term scheduler strives for good **process mix**



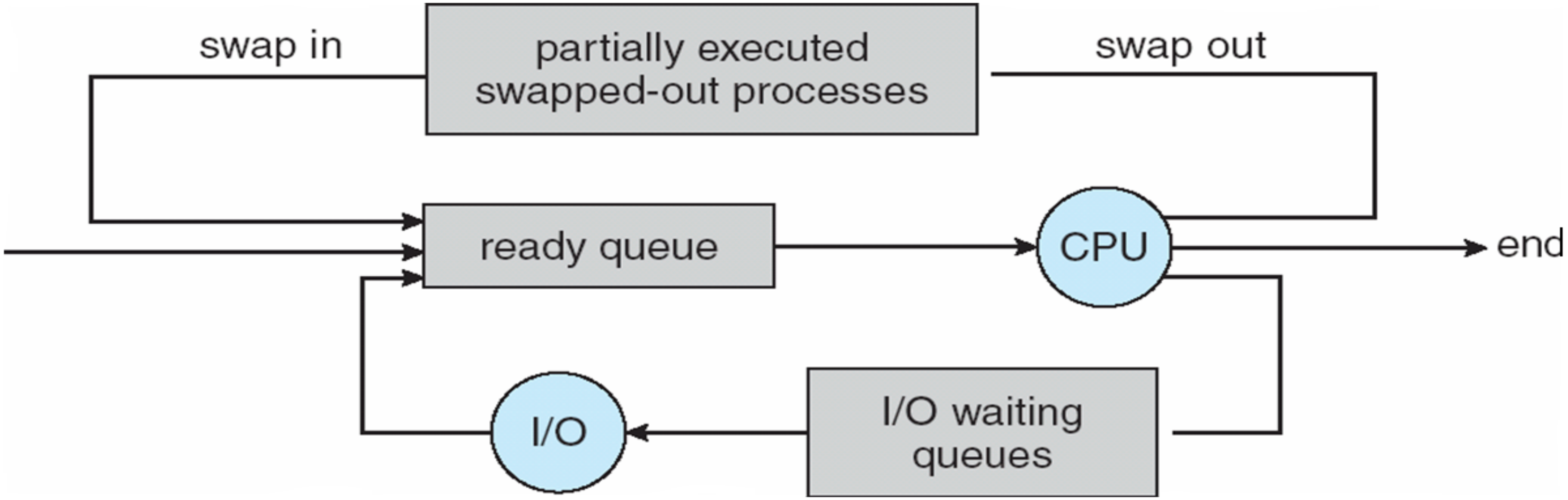
- Remarks：
  - Control the degree of multiprogramming
  - Can take more time in selecting processes because of a longer interval between executions
  - May not exist physically

Short-Term Scheduler

- Goal: To efficiently allocate the CPU to one of the ready processes according to some criteria
- Short-term scheduler is invoked very frequently (milliseconds) ➔ must be fast
- In Linux, after version 2.6.23, the scheduler is the Completely Fair Scheduler (CFS)

Medium-Term Scheduler

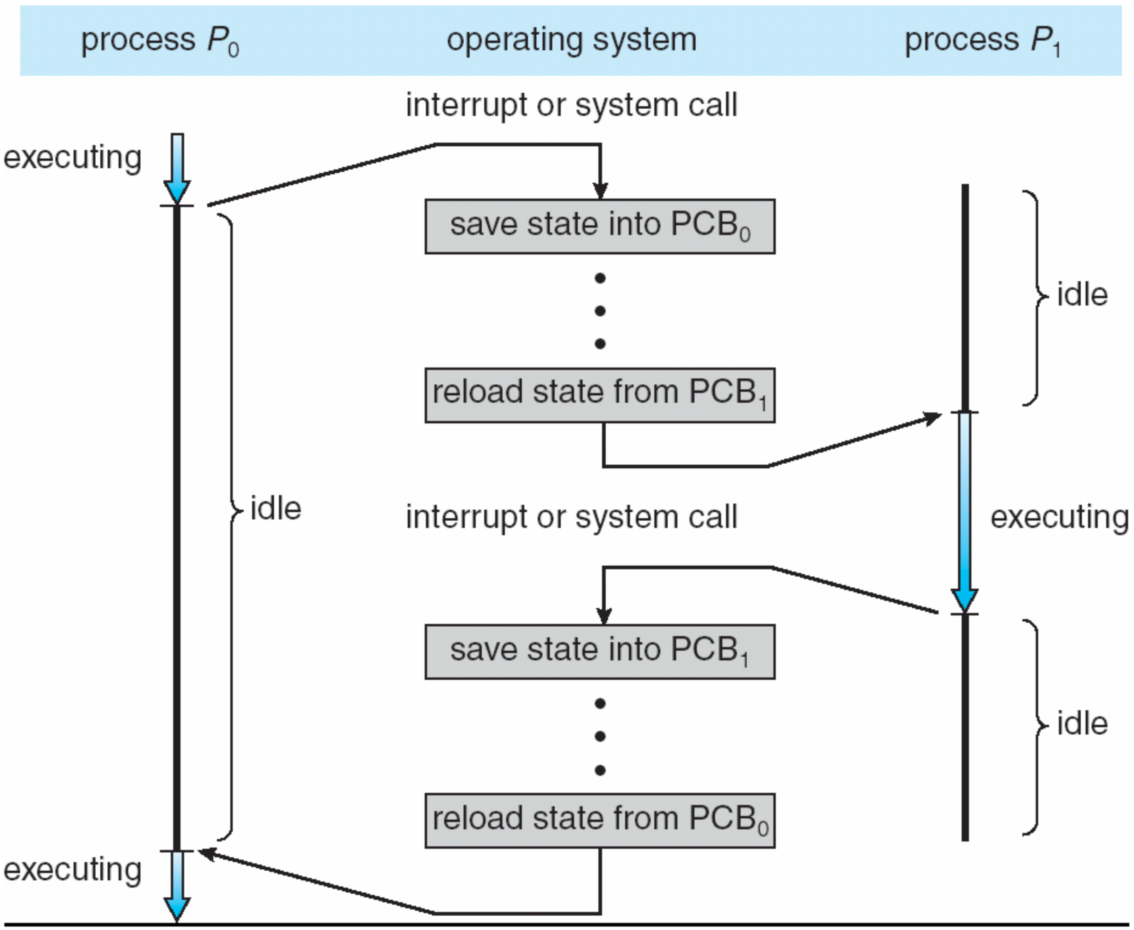
- Goal: Remove process from memory, store on disk, bring back in from disk to continue execution: it is also called “swapping”



Process Scheduling– Context Switches

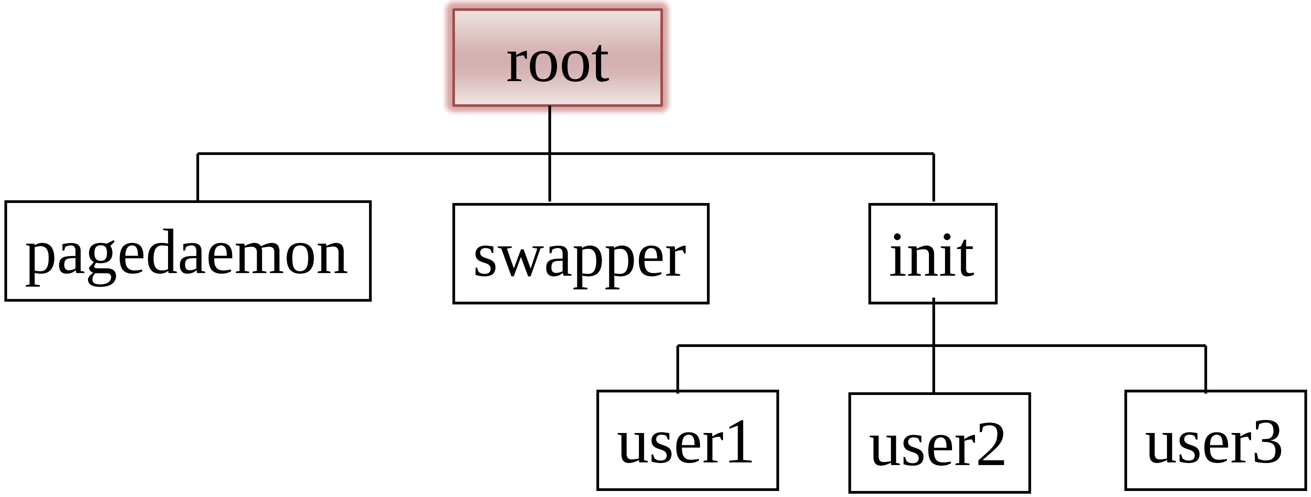
- Context Switch: Pure Overheads
  - Save the state of the old process and load the state of the newly scheduled process.
    - The context of a process is usually reflected in PCB
- Issues:
  - The cost depends on the hardware support
    - e.g. processors with multiple register sets or computers with advanced memory management
  - Threads, i.e., light-weight process (LWP), are introduced to break this bottleneck

CPU Switch from Process to Process



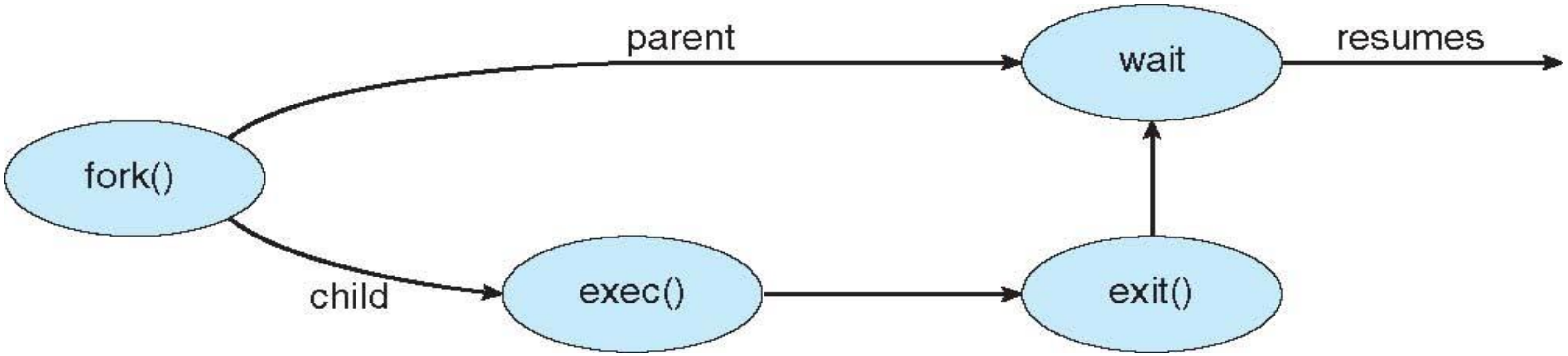
Parent and Child Processes

- Parent processes create child processes, which in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (PID)



Process Creation

- Address Space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX Examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



Process Termination

- Process executes last statement and asks the operating system to delete it: **`exit()`**
  - Wait the output data from child to parent: **`wait()`**
- Parent may terminate the execution of child processes: **`abort()`**

  ➔Child has exceeded allocated resources

  ➔Task assigned to child is no longer required
  - Receive the return value form child
  - Some operating systems do not allow child to continue if its parent terminates
    - All children should be terminated - **cascading termination**

C Program Forking a Process

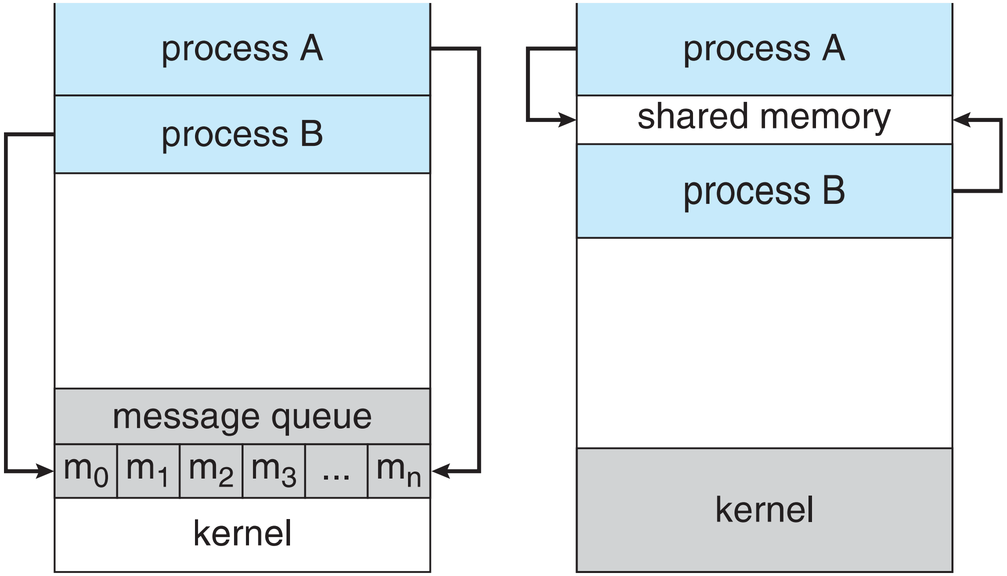
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 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");
    }

    return 0;
}
```

Inter-Process Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need inter-process communication (IPC)

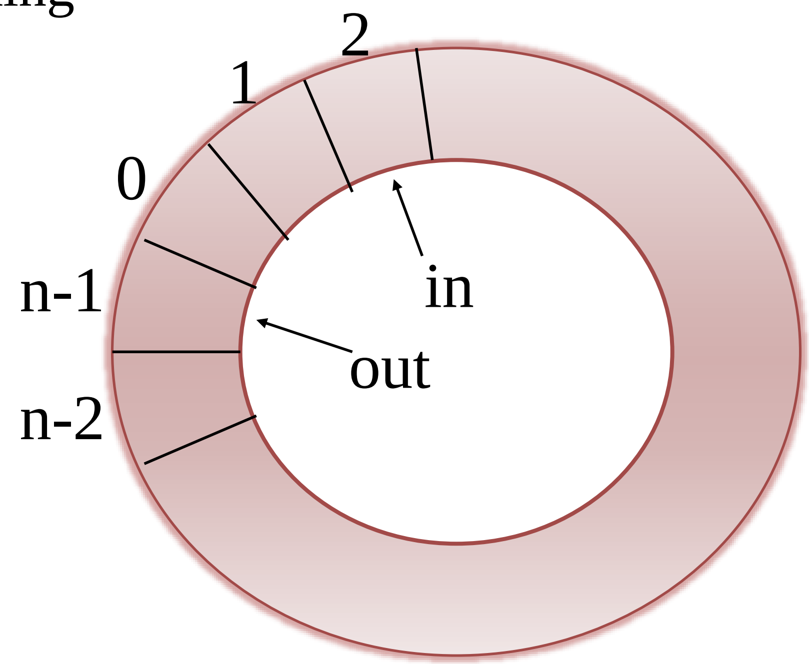
Two Models of IPC

- Shared Memory
  - Max Speed & Communication Convenience
- Message Passing
  - No Access Conflict & Easy Implementation



Shared Memory IPC

- A Consumer-Producer Example:
  - Bounded buffer or unbounded buffer
    - Supported by inter-process communication (IPC) or by hand coding



buffer[0…n-1]

Initially, in=out=0

Shared Memory– Consumer

```
while (true)
{
   while (in == out);
   /* do nothing and have to wait */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   … /* use the consumed item */
}
```


Shared Memory– Producer

```
while (true)
{
  … /* produce a new item */
  while (((in + 1) % BUFFER_SIZE) == out);
  /* do nothing */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER SIZE;
}
```


Message Passing IPC

- Logical Implementation of Message Passing
  - Fixed/variable message size
  - Symmetric/asymmetric communication
  - Direct/indirect communication
  - Synchronous/asynchronous communication
  - Automatic/explicit buffering
  - Send by copy or reference

Direct Message Passing

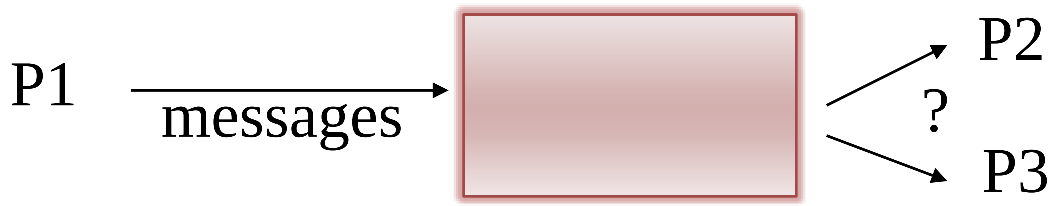
- Processes must name each other explicitly:
  - **send** (*P*, *message*) – send a message to process P
  - **receive**(*Q*, *message*) – receive a message from process Q
- Properties of the communication link
  - Links are established automatically
  - A link is associated with exactly one pair of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

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - **`send`**(*A*, *message*) – send a message to mailbox A
  - **`receive`**(*A*, *message*) – receive a message from mailbox A
- Properties of the communication link
  - Links are established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Links may be unidirectional or bi-directional

Issues of 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

IPC Synchronization

- Synchronous Message Passing IPC
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- Asynchronous Message Passing IPC
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null

IPC Buffering

- The capacity of a link: the number of messages could be held in the link
  - Zero capacity – 0 messages
    - Sender must wait for receiver
  - Bounded capacity – finite length of n messages
    - Sender must wait if link is full
  - Unbounded capacity – infinite length
    - Sender never waits
- The last two items are for asynchronous communication and may need acknowledgement

Examples of IPC Systems – POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment

```
 shm_fd = shm_open(name, O_CREAT | O_RDWR,
0666);
```

  - Set the size of the object

```
ftruncate(shm_fd, 4096);
```

  - Memory map the object

```
ptr = mmap(0,4096,PROT_WRITE,MAP_SHARED,
    shm_fd, 0);
```

  - Now the process could write to the shared memory

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


Examples of IPC Systems – Mach

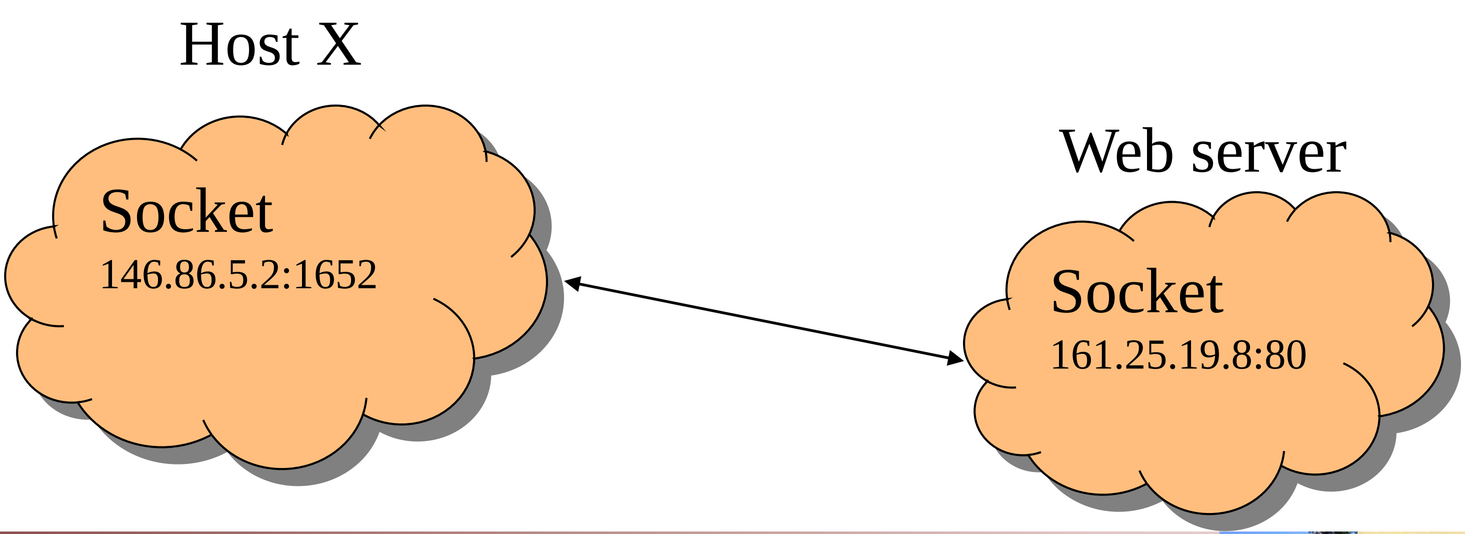
- Mach – A message-based OS from the Carnegie Mellon University
  - When a task is created, two special mailboxes, called ports, are also created.
    - The *Kernel* mailbox is used by the kernel to communicate with the tasks
    - The *Notify* mailbox is used by the kernel sends notification of event occurrences.

Three IPC System Calls on Mach

- msg_send
  - Options when mailbox is full:
    - Wait indefinitely
    - Return immediately
    - Wait at most for n ms
    - Temporarily cache a message: only one message to a full mailbox can be pending at any time for a sending tread
- msg_receive
  - Only one task can own & have a receiving privilege of a mailbox
  - Options when mailbox is empty:
    - Wait indefinitely
    - Return immediately
    - Wait at most for n ms
- msg_rpc
  - Remote Procedure Calls

Communication in Client-Server Systems

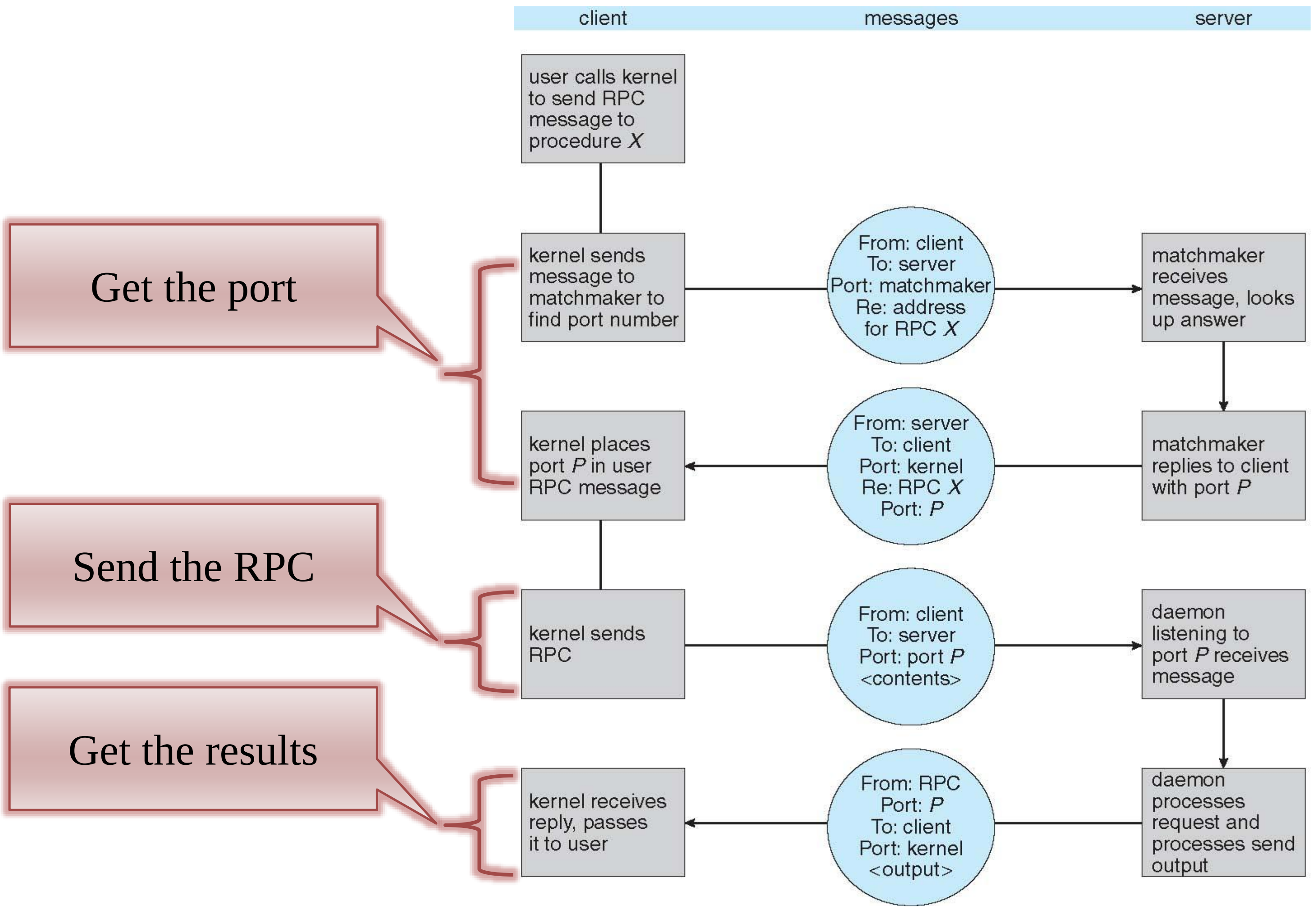
- Socket
  - An endpoint for communication identified by an IP address concatenated with a port number
    - A client-server architecture
- /etc/services: 23-telnet, 21-ftp, 80-web server, etc.



Remote Procedure Calls

- A way to abstract the procedure-call mechanism for use between systems with network connection
- Stubs at the client site
  - One for each RPC
  - Locate the proper port and marshal parameters
- Stubs at the server site
  - Receive the message
  - Invoke the procedure and return the results
- Data representation handled via the External Data Representation (XDL) format to account for different architectures
  - Big-endian and little-endian

Execution of RPC

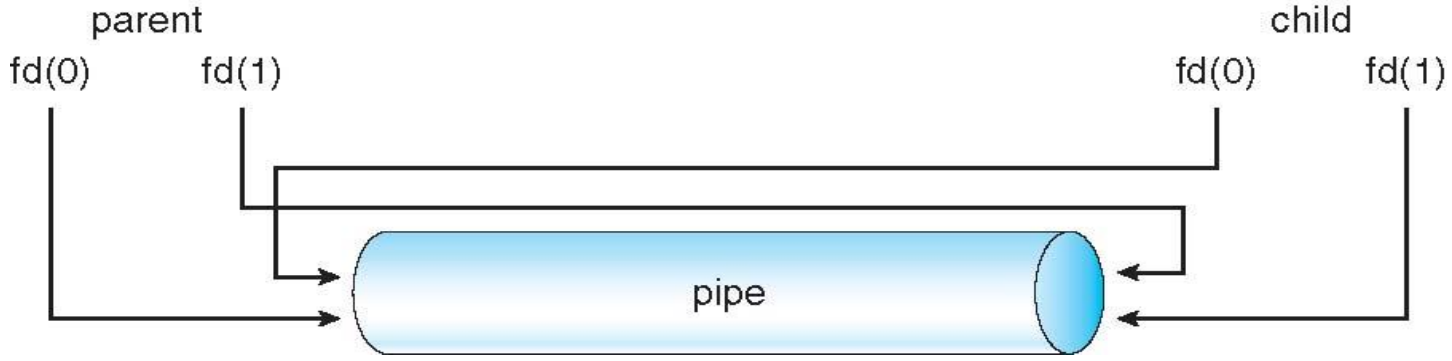


Pipes

- Acts as a conduit allowing two processes to communicate
- Issues
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half-duplex or full-duplex?
  - Must there exist a relationship (i.e. parent-child) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary Pipes

- Ordinary Pipes allow communication in the standard producer-consumer style
- Producer writes to the write-end of the pipe
- Consumer reads from the read-end of the pipe
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems