Chapter 3 - Process

Date created	@November 9, 2022
Source	Slides
Type	Slide

Process

▼ what is a process?

- a program in execution
- **process execution must progress in sequential fashion**

▼ how are all processes executed?

Concurrently

▼ what is the difference between a program and a job?

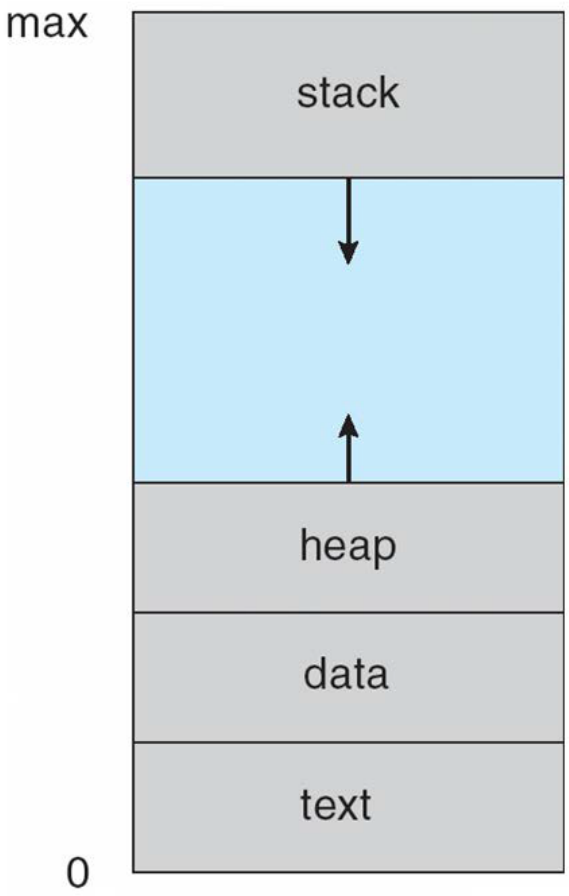
- Program —> Passive
- Process —> Active

 Program becomes process when executable file loaded into memory

 One program can be several processes

🎓 for example, a java program is not a process because its a .java file but when its loaded on memory, converted to binary and assembly code, and starts executing, it is considered as a process

Process in memory



▼ what is stored in **Text section**?

the executable code, this is read-only and might be shared by a number of processes

▼ what is stored in **Data section**?

global variables

▼ what is stored in **Heap section**?

memory dynamically allocated during the program run time

▼ what is stored in **Stack section?**

temporary data (such as function parameters, return addresses, and local variables)

Process state

▼ what is process state?

while executing, a process may have different states and it may change state at some certain point based on some certain events.

▼ what are the different states of a process?

- **New**: the process is being created
- **Running**: instructions are being executed
- **Waiting**: the process is waiting for some event to occur (such as an I/O completion or reception of a signal)
- **Ready**: the process is waiting to be assigned to a processor
- **Terminated**: the process has finished execution.

Process control block (PCB)

▼ what is a process control block?

For each process there is a Process Control Block(PCB) – also called a **task control block**

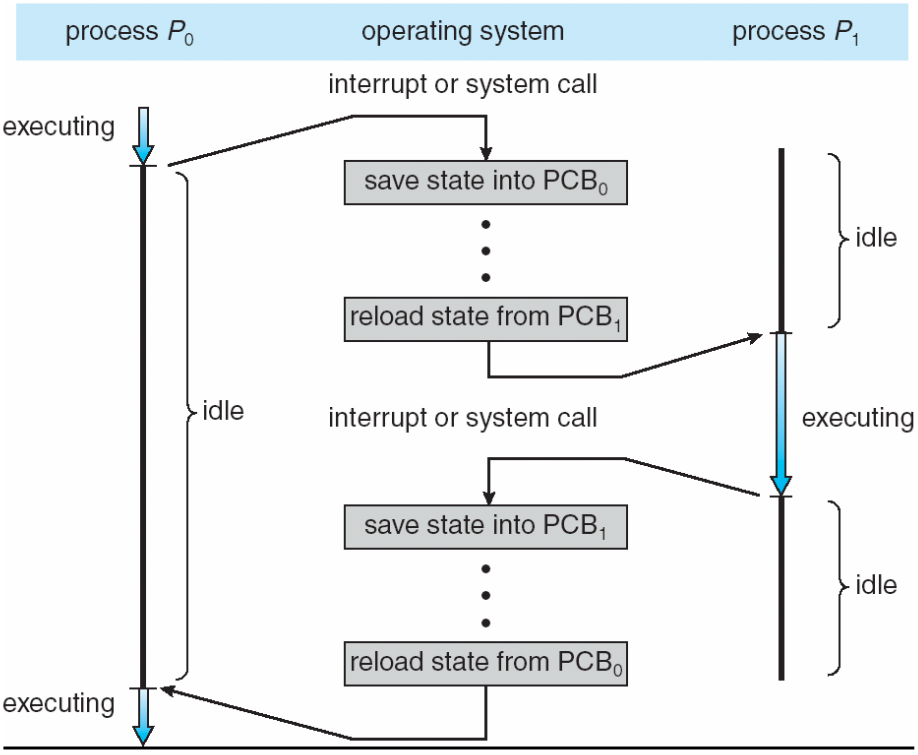
▼ what is stored inside PCB?

Information associated with each process

▼ what are the information stored inside PCB?

- **Process State** – running, waiting, etc., as discussed above
- **Process Number "PID"** – is a number used by most operating system kernels to uniquely identify an active process
- **Program counter** – the counter indicates the address of the next instruction to be executed for this process
- **CPU Registers** – tell us the particular registers that are being used by a particular process
- **CPU-Scheduling information** – this information includes a process priority, pointers to scheduling queues, and any other scheduling parameters
- **Memory-Management information** – e.g., page tables or segment tables, depending on the memory system used by the operating system
- **Accounting information** – they are the resources that are used by a specific process e.g., the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on
- **I/O Status information** - e.g., a list of I/O devices allocated to the process, a list of open files, and so on.

CPU switches from process to process



▼ how exactly does cpu execute processes **“concurrently”?**

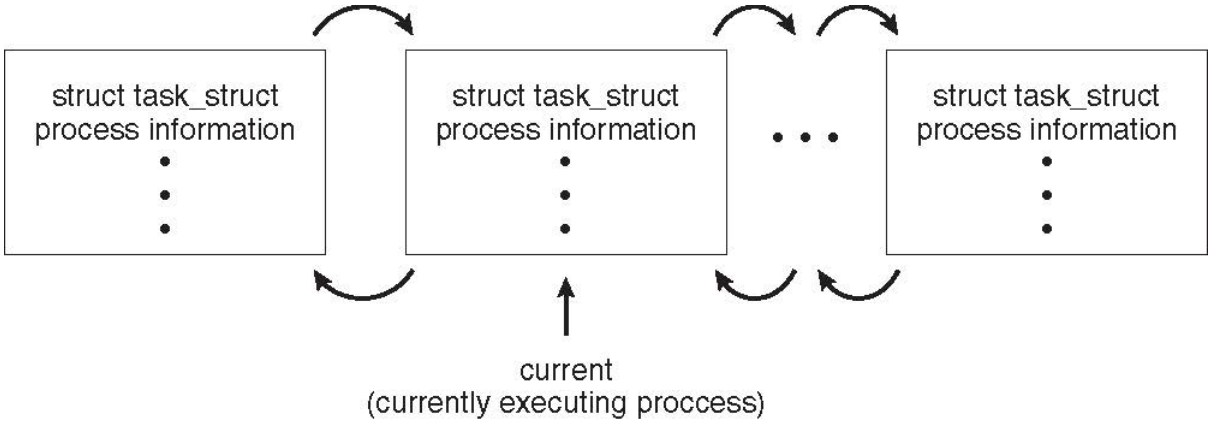
it starts a process and switches to another one frequently.

to switch from one process to another it has to switch contexts of PCB’s

As you can see the time that takes cpu to switch from two processes, it is actually saving latest info on PCB0 and reloading latest info from PCB1. meanwhile both of Processes are idle in **WAIT** state. this time is called OS overhead

Process representation in Linux

```
pidt_pid; /* process identifier */
long state; /* state of the process */
unsigned inttime_slice/* scheduling information */
structtask_struct*parent; /* this process's parent */
structlist_headchildren; /* this process's children */
structfiles_struct*files; /* list of open files */
structmm_struct*mm; /* address space of this process */
```

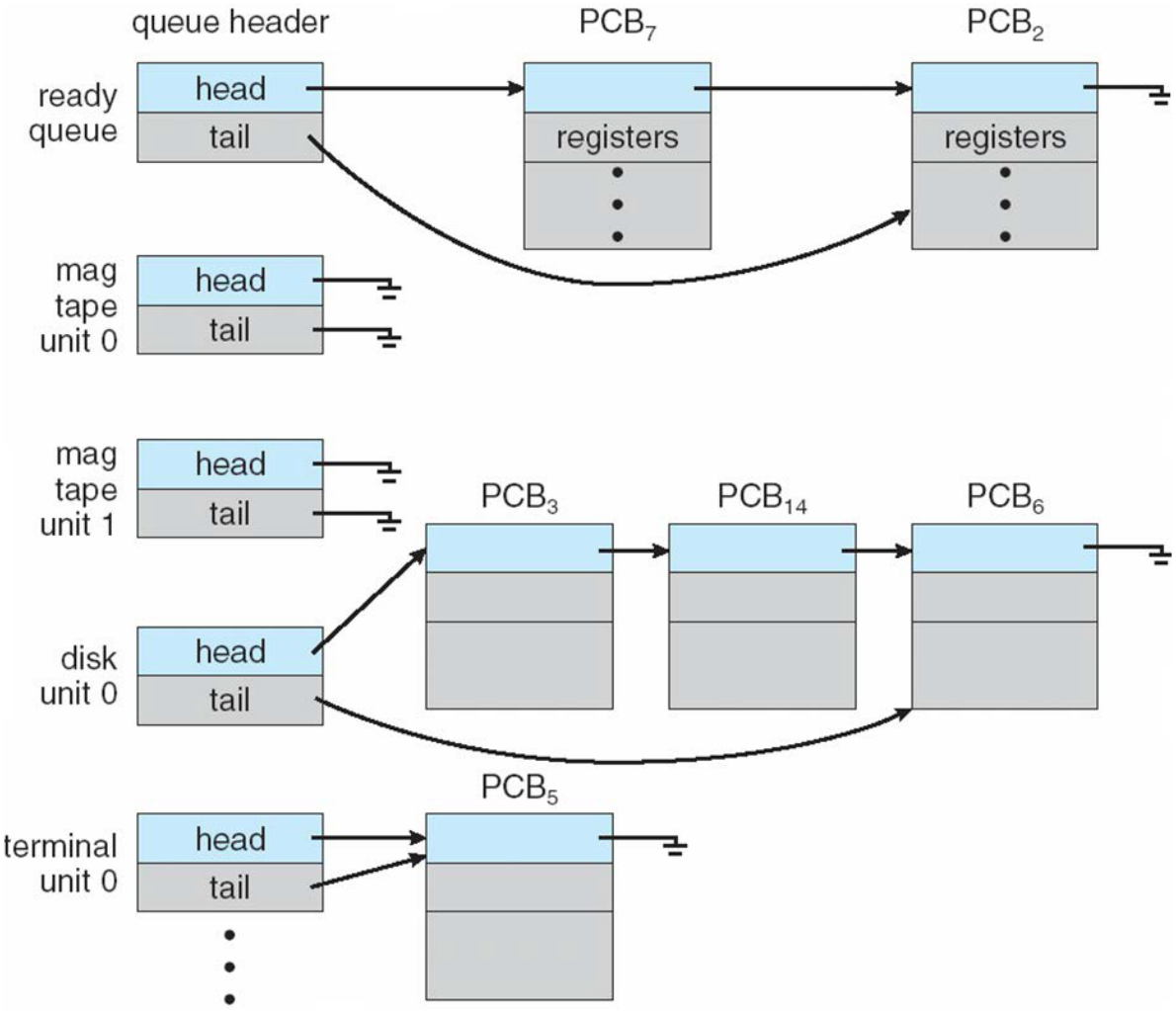


🎓 as you can see, processes are stored in a linked list

Process scheduling

▼ what does process scheduler do?

Process scheduler selects among available processes for next execution on CPU



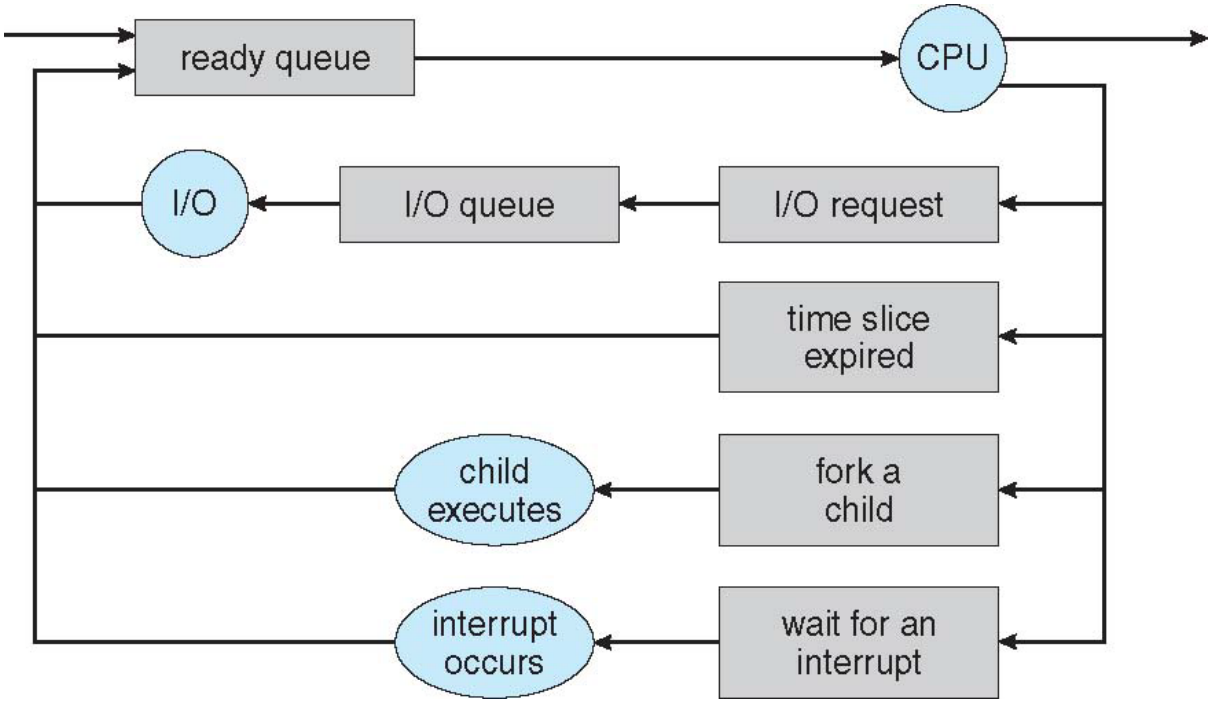
▼ name every scheduling queues of processes?

- Job queue – set of all processes in the system
- Ready queue – set of all processes residing in main memory, ready and waiting to execute
- Device queues – set of processes waiting for an I/O device

> 🎓 processes migrate among the various queues

Diagram representation of process scheduling

Queueing diagram represents queues, resources, flows



Schedulers

▼ what is Short-term scheduler?

- aka CPU scheduler
- selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒(must be fast)

▼ what is Long-term scheduler?

- aka Job scheduler
- selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒(may be slow)

- The long-term scheduler controls the degree of multiprogramming

▼ How can Processes be described?

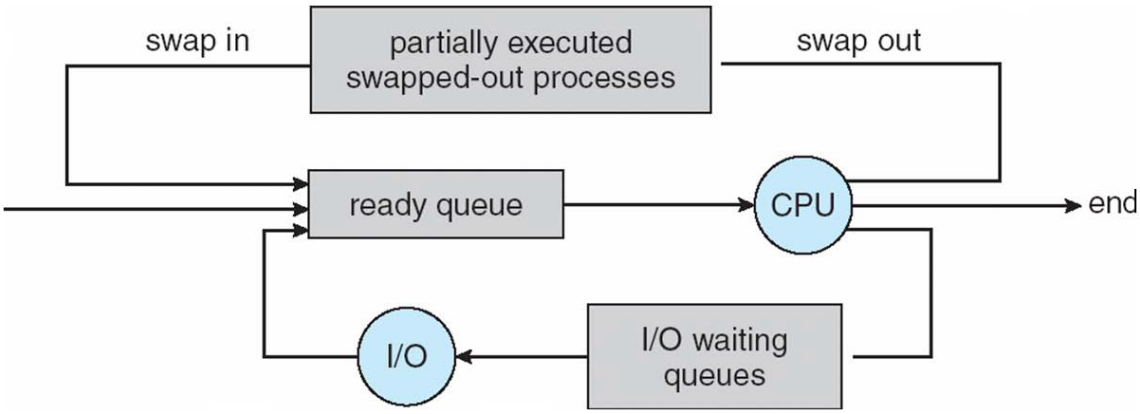
there are 2 ways to describe a Process:

- **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

▼ what is medium term scheduler?

- can be added if degree of multiple programming needs to decrease
- Remove process from memory, store on disk, bring back in from disk to continue execution: swapping



Context switch

▼ what is 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

▼ where is the context of a process stored?

the context of a process is actually the info stored in PCB

▼ what is an overhead?

Context-switch time is overhead; the system does no useful work while switching

> 🎓 The more complex the OS and the PCB ⇒ the longer the context switch

> 🎓 Some hardware provides multiple sets of registers per CPU ⇒ multiple contexts loaded at once

Process creation

processes can exist by running a program

or can be made by some code execution inside the code running in other process.

the new process created is called **Child** process and the original process is the **Parent** process.

Parent vs. **Child**

> 🎓 Generally, process identified and managed via a process identifier (pid)

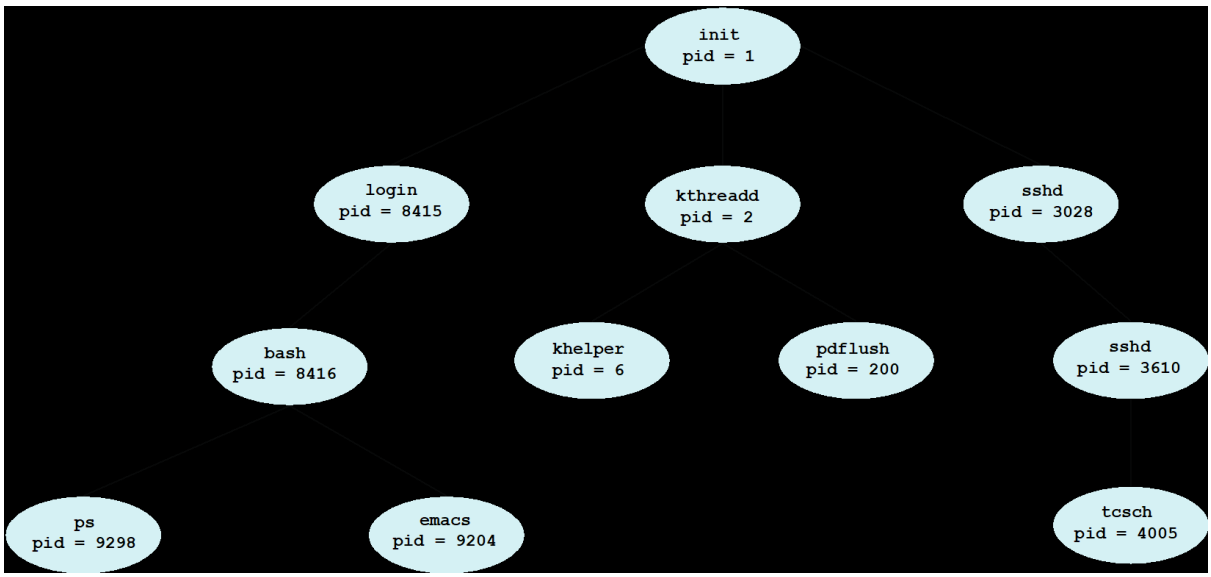
▼ what are the resource sharing options among **Parent** and **Child**?

- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources (this method is used in **windows**)

▼ what are the execution options among **Parent** and **Child**?

- Parent and children execute concurrently

- Parent waits until children terminate



A tree of processes in Linux

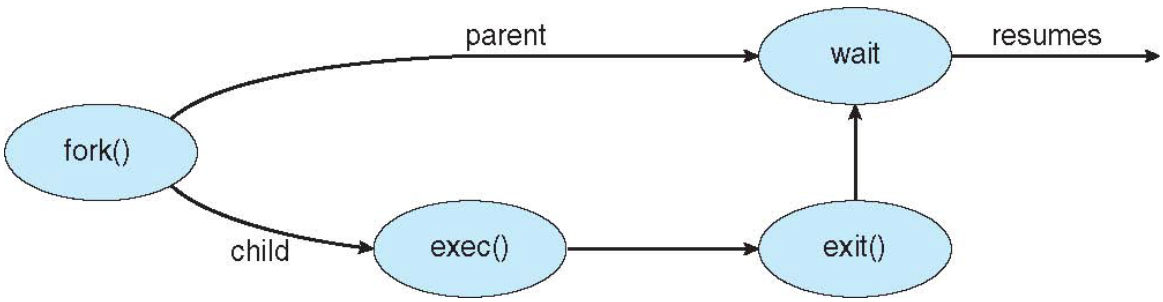
what are the address space options among **Parent** and **Child**?

- Child duplicate of parent
- Child has a program loaded into it

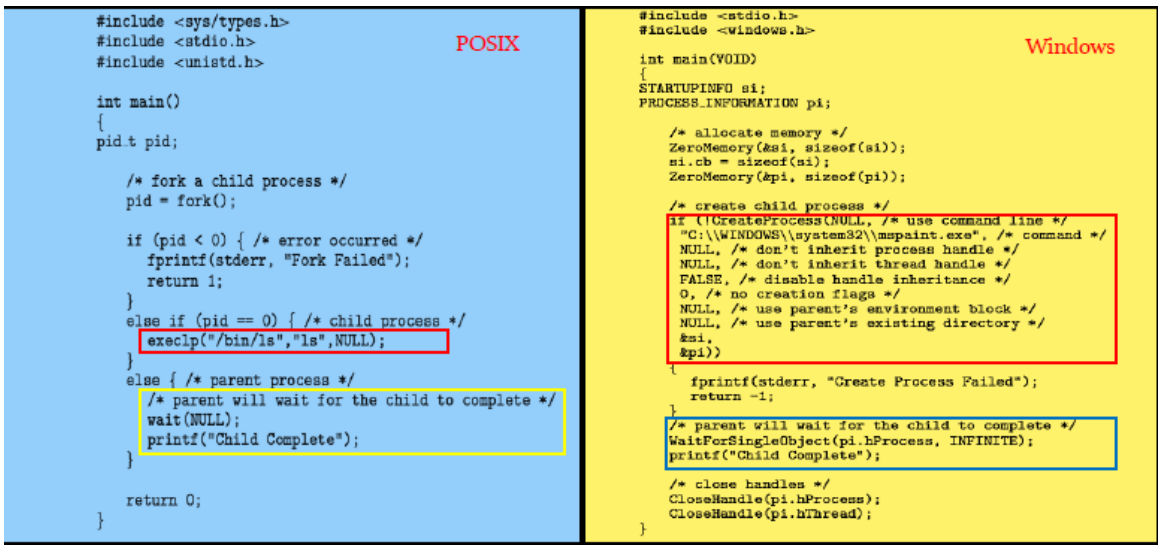
it is comparing of having the same app or a new app opened as a process

some 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 creation with C



Process termination

▼ how the termination of a **Child** process is terminated?

Process executes last statement and then asks the operating system to delete it using the exit() system call. (Also return() but its used in main()).

- Returns status data from child to parent (via wait())
- Process' resources are deallocated by operating system

▼ How the **Parent** can terminate **Child** process?

Parent may terminate the execution of children processes using the abort() system call.

▼ Why **Parent** terminates **Child** process ?

- Child has exceeded allocated resources.
- Task assigned to Child is no longer required.
- The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Problems of process termination

▼ what is a **zombie process**?

- If no parent waiting (did not invoke wait()) process is a zombie

▼ what is an **orphan process**?

- If parent terminated without invoking wait , process is an orphan

Multi process example : Chrome Browser

each tab is considered as a process (when a page server crash it does not affect other pages, cause its a different process than others).

Interprocess communication (IPC)

▼ how can the processes within a system be?

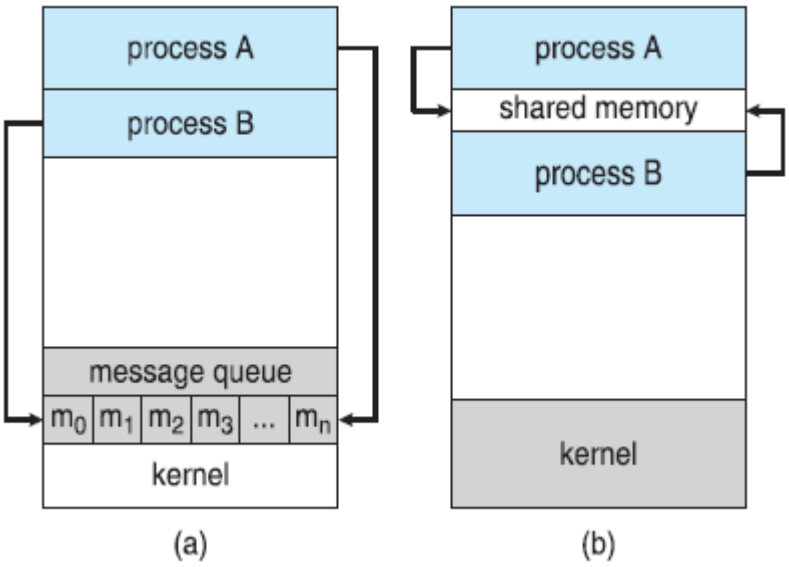
Processes within a system may be **independent** or **cooperating**

> 🎓 Cooperating process can affect or be affected by other processes, including sharing data

> 🎓 cooperating processes, are usually faster

▼ what are different types of Cooperating processes?

- Shared memory
- Message passing



(a) Message passing. (b) shared memory.

Circular buffer & producer-consumer problem

study slide

Message passing

▼ what are two different communication links in message passing?

- Direct communication (unidirectional)
- Indirect communication (uni & bidirectional)

▼ what are the functions in Direct Communication?

- send(P, message) –send a message to process P
- receive(Q, message) – receive a message from process Q

▼ How does indirect communication work?

- Messages are directed and received from mailboxes (also referred to as ports)
- Each mailbox has a unique id

- Processes can communicate only if they share a mailbox

▼ what are the functions in Indirect communication?

- send(A, message) – send a message to mailbox A
- receive(A, message) – receive a message from mailbox A

Synchronization

▼ what are two different types of message passing?

- blocking
- non-blocking

▼ what is **Blocking** message passing?

- considered as Synchronous
- Blocking send - the sender is blocked until the message is received
- Blocking received - the receiver is blocked until a message is available

▼ what is **non-Blocking** message passing?

- considered as Asynchronous
- Non-blocking send -- the sender sends the message and continue
- Non-blocking receive -- the receiver receives:
  - A valid message

  or
  - Null message

If both send and receive are blocking, we have a **rendezvous**

Example of sender & receiver in shared mem

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

Sender

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Receiver

Communication in client-server systems

- Sockets
- Remote procedure calls (its a windows ability - you can call procedures on another system)

- Pipes
- remote method Invocation

Sockets

▼ what is a socket?

A socket is defined as an endpoint for communication

▼ what is the use of concatenation of IP address and port?

a number included at start of message packet to differentiate network services on a host

The socket 161.25.19.8:1625refers to port 1625on host 161.25.19.8

▼ what is the requirement of establishing a communication?

Communication consists between a pair of sockets

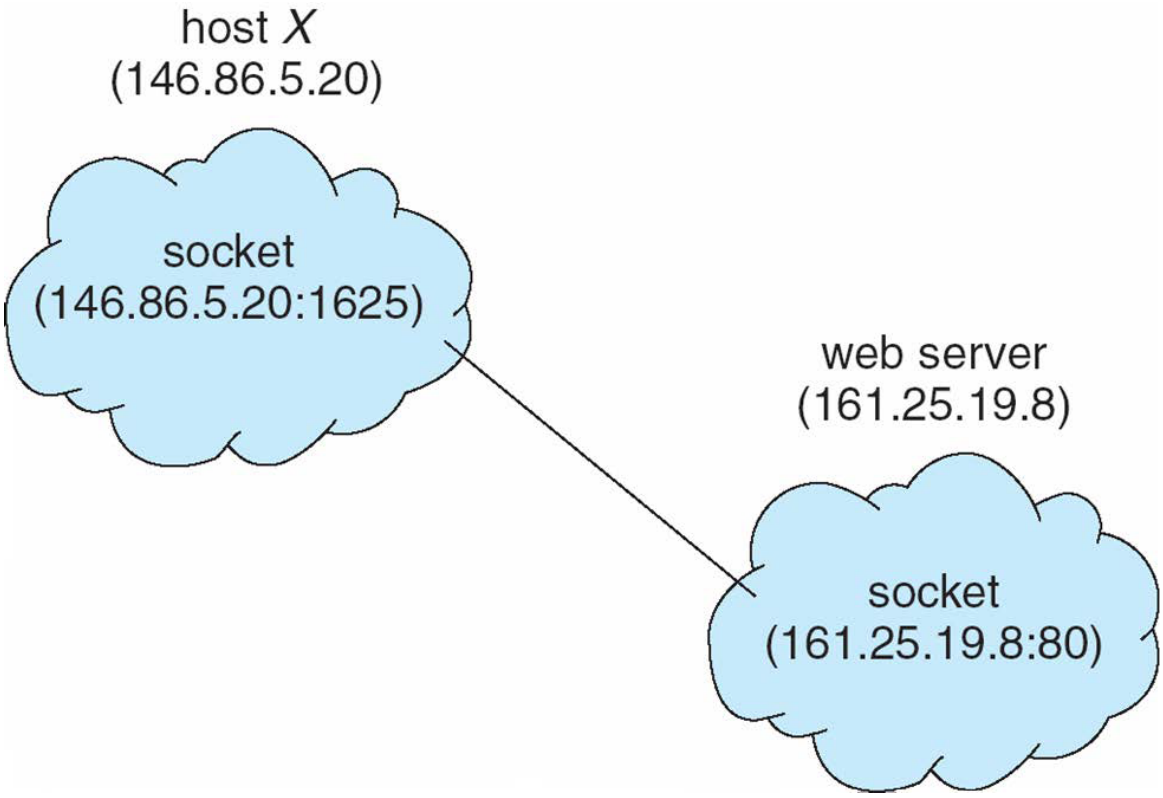
▼ what ports are considered **well known**?

All ports below 1024 are well known, used for standard services

▼ what is IP address 127.0.0.1 ?

Special IP address 127.0.0.1 (loopback) to refer to system on which process is running

Example of socket communication

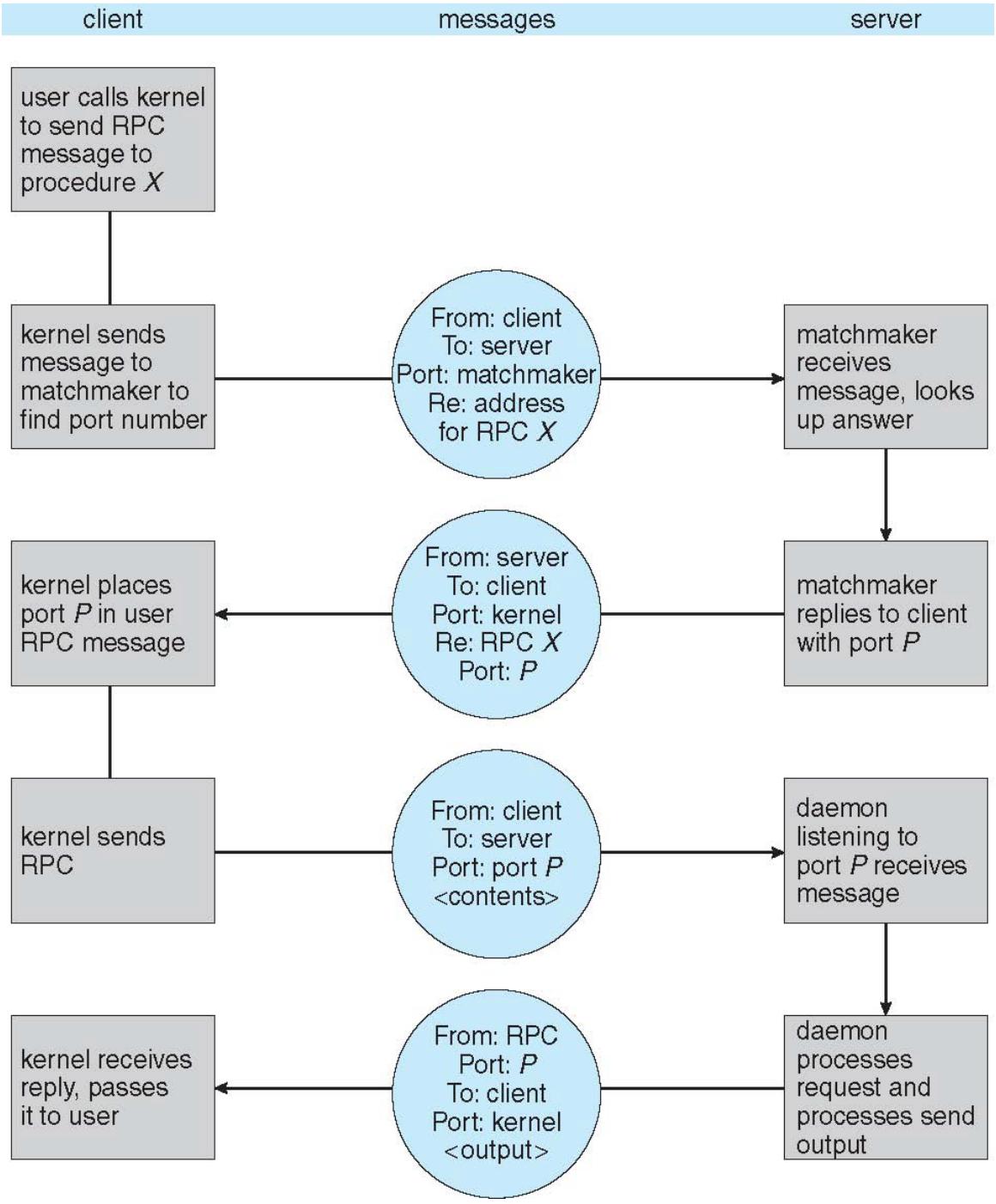


▼ what are the 3 types of sockets in JAVA?

- Connection-Oriented (TCP)
- Connectionless (UDP)
- MulticastSocket class–data can be sent to multiple recipients

Remote Procedure Call (RPC)

This is how the execution of RPC is done:



Pipes

▼ what is a pipe?

Acts as a conduit allowing two processes to communicate

🎓 conduit = مجرا، لوله

▼ Issues on pipes

- Is communication unidirectional or bi-directional?
- In the case of two-way communication, is it half or full-duplex?

  half = one procedure talks, the other listen

  full-duplex = both of procedures talk and listen at the same time
- Must there exist a relationship (i.e., parent-child) between the communicating processes?
- Can the pipes be used over a network?

▼ what are the different type of pipes?

- **Ordinary pipes**

  cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.

  🎓 only parent-child relation is supported

- **Named pipes**

  can be accessed without a parent-child relationship.

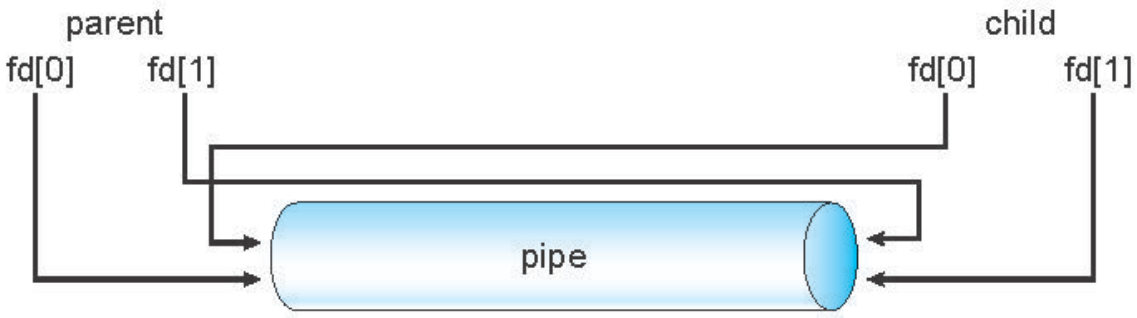
  🎓 can have connections to any pipes - also it can be more than a binary relationship.

Ordinary Pipes

▼ how does ordinary pipe work?

Ordinary Pipes allow communication in standard producer-consumer style

- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore **unidirectional**
- Require **parent-child relationship** between communicating processes

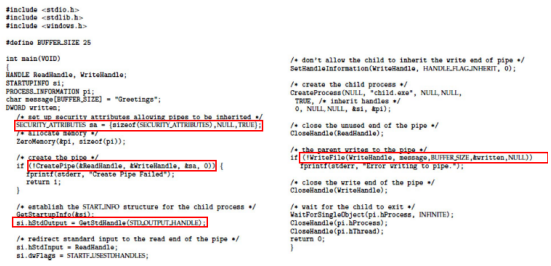


Windows calls these **anonymous pipes**

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

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;
    /* create the pipe */
    if (pipe(fd) == -1) {
       fprintf(stderr,"Pipe failed");
       return 1;
    }

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

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
```

```c
    if (pid > 0) { /* parent process */
       /* close the unused end of the pipe */
       close(fd[READ_END]);

       /* write to the pipe */
       write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

       /* close the write end of the pipe */
       close(fd[WRITE_END]);
    }
    else { /* child process */
       /* close the unused end of the pipe */
       close(fd[WRITE_END]);

       /* read from the pipe */
       read(fd[READ_END], read_msg, BUFFER_SIZE);
       printf("read %s",read_msg);

       /* close the write end of the pipe */
       close(fd[READ_END]);
    }

    return 0;
}
```

Ordinary pipe (POSIX), parent-child



```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char message[BUFFER_SIZE] = "Greetings";
DWORD written;
  /* set up security attributes allowing pipes to be inherited */
  SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
  /* ALLOCATE MEMORY */
  ZeroMemory(&pi, sizeof(pi));

  /* create the pipe */
  if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
    fprintf(stderr, "Create Pipe Failed");
    return 1;
  }

  /* establish the START_INFO structure for the child process */
  GetStartupInfo(&si);
  si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

  /* redirect standard input to the read end of the pipe */
  si.hStdInput = ReadHandle;
  si.dwFlags = STARTF_USESTDHANDLES;

  /* don't allow the child to inherit the write end of pipe */
  SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

  /* create the child process */
  CreateProcess(NULL, "child.exe", NULL, NULL,
    TRUE, /* inherit handles */
    0, NULL, NULL, &si, &pi);

  /* close the unused end of the pipe */
  CloseHandle(ReadHandle);

  /* the parent writes to the pipe */
  if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
    fprintf(stderr, "Error writing to pipe.");

  /* close the write end of the pipe */
  CloseHandle(WriteHandle);

  /* wait for the child to exit */
  WaitForSingleObject(pi.hProcess, INFINITE);
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);
  return 0;
}
```

Ordinary pipe (windows), parent

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

   /* get the read handle of the pipe */
   ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

   /* the child reads from the pipe */
   if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
     printf("child read %s",buffer);
   else
     fprintf(stderr, "Error reading from pipe");

   return 0;
}
```

Ordinary pipe (windows), child

Named Pipes

▼ comparing to Ordinary pipes

Named Pipes are more powerful than ordinary pipes, reason:

- there is no need for parent-child relation

- Communication is **bi-directional**
- 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