CS 5523 Operating Systems: Process Management

Instructor: Dr. Tongping Liu

Thank Dr. Dakai Zhu, Dr. Palden Lama, and Dr. Tim Richards (UMASS) for providing their slides.
Outline

- Basic concepts of process
  - Address space and Process control block (PCB)

- Basic operations for process management
  - Process creation/termination

- Scheduling of process: CPU scheduling
  - Basic scheduling algorithms: FIFO, SJF etc

- Inter process communication (IPC)
  - shared memory vs. message passing
Process vs. Program

- **Program**: a set of functions
  - a *passive* entity
  - stored as files on disk

- Process: a *program in execution*
  - Dynamic concept: running of a program

How do we run a program?
What are steps to create a process?
Address Space in Process

- Range of memory locations
  - code section
  - data section
  - Stack
  - Auxiliary: environment variables and command line arguments
  - Heap: memory for dynamically allocated data items

How big is the address space?
As a process executes, it changes 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
Transitions of Processes

- Transitions between states
  1 - Process enters ready queue
  2 - Scheduler picks this process
  3 - Scheduler picks a different process
  4 - Process waits for event (such as I/O)
  5 - Event occurs
  6 - Process exits
  7 - Process ended by another process
Process: Running Context

- **Registers**: in addition to general registers
  - **Program Counter (PC)**: contains the memory address of the next instruction to be executed.
  - **Stack Pointer (SP)**: points to the top of the current stack in memory. The stack contains one frame for each procedure that has been entered but not yet exited.
  - **Program Status Word (PSW)**: is an IBM System/360 architecture and successors control register, which performs the function of a Status register and Program counter in other architectures.

- Higher level resources: open files etc.
- Synchronization and communication resource: semaphores and sockets
Process Control Block (PCB)

- OS creates a PCB for each process
- OS manages the processes → a list of PCBs

<table>
<thead>
<tr>
<th>process state</th>
</tr>
</thead>
<tbody>
<tr>
<td>process number</td>
</tr>
<tr>
<td>program counter</td>
</tr>
<tr>
<td>registers</td>
</tr>
<tr>
<td>memory limits</td>
</tr>
<tr>
<td>list of open files</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Example: PCB in Linux (task_struct)

- **Process Management**
  - Registers
  - Program Counter
  - Stack Pointers
  - Process State
  - Priority
  - Scheduling Parameters (slice)
  - Process ID
  - Parent process
  - Process group
  - Time when process started
  - CPU time used

- **Memory Management**
  - Pointer to text (code) segment
  - Pointer to data segment
  - Pointer to stack segment

- **File Management**
  - Root directory
  - Working directory
  - User Id
  - Group Id
  - List of open files
PCB is Used in Context Switches
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 overhead
  - 1 ~ 1000 ms

- Hardware support
  - Multiple set of registers

- Other performance issues/problems
  - Cache content: locality is lost
  - TLB content: may need to flush
Outline

- Basic concepts of process
  - Address space and Process control block (PCB)
- Basic operations for process management
  - Process creation/termination
- Scheduling of process: CPU scheduling
  - Basic scheduling algorithms: FIFO, SJF etc
- Inter process communication (IPC)
  - shared memory vs. message passing
Process Creation and Hierarchy

- **When to create processes?**
  - System initialization
  - User request to create a new process
  - Running processes use system call to create new process (exec() family)

- **Process hierarchy**
  - Parent process creates child process(es)

**Process Tree in Solaris**
How do we control processes?
Process Creation

- **Address space**
  - Child duplicate its parent (data, program)
  - Child has a program loaded into it (exec())

- **Execution**
  - Parent and children execute concurrently
  - Parent waits until children terminate

- **Resource sharing: complicated**
  - Shared open files, but not descriptors
  - Different page tables
Process Creation: System Call in LINUX

- Each process has a process identifier (pid)
- Parent process executes `fork` to spawn a child process
- The child process has a separate copy of the parent’s address space.
- Both the parent and the child continue execution at the instruction following the `fork` system call
  - Return value of 0 → new (child) process continues
  - Otherwise, pid of child process → parent process continues
An Example: Unix `fork()`

```c
int cpid = fork();
if (cpid == 0) {
    <child code>
    exit(0);
}<parent code>
waitpid(cpid);
```

```c
<...>
int cpid = fork();
if (cpid == 0) {
    <child code>
    exit(0);
}<parent code>
waitpid(cpid);
<...>
```
Load A Different Program

Child process uses `execlp` to load a different program

```c
void main (){  
    int pid;  
    pid = fork();  
    if (pid < 0) {error_msg}
    else if (pid == 0) {/* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for
         * parent will wait for
        wait(NULL);
        exit(0);
    }
}
void main()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}

What does this print out?

```
L0
L1
L1
Bye
Bye
Bye
Bye
```
Create & Terminate Processes

What does this print out?

```c
void main()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

Does it always print in order?
Create & Terminate Processes

What does this print out?

```c
void main()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```

05_fork.c
Create & Terminate Processes

What happens here?

```c
void main()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid);
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid);
        while (1) ; /* Infinite loop */
    }
}
```

05_zombie.c

Zombie!
05_zombie.c

Linux

[elnuxl:Exceptions and Processes Part 1 Code] ./05_zombie
Running Parent, PID = 20201
Terminating Child, PID = 20202
^Z
[1]+ Stopped ./05_zombie
[elnuxl:Exceptions and Processes Part 1 Code] ps
  PID  TTY TIME CMD
19886 pts/1  00:00:00 bash
20201 pts/1  00:00:01 05_zombie
20202 pts/1  00:00:00 05_zombie <defunct>
20204 pts/1  00:00:00 ps
void main()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        execv("/bin/ls", "ls");
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1); /* Infinite loop */
    }
}
Create & Terminate Processes

What about this one?

void main()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n", 
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n", 
               getpid());
        exit(0);
    }
}

06_zombie.c
06_zombie.c

Really hard to detect since the main process already exited!
How do we control processes?

Need a way to kill the zombies!

1. Choose your weapon
2. Aim for the head
3. Don't miss (or it will eat your brains)
Need a way to kill the zombies!

This is called reaping!

How do we control processes?
Create & Terminate Processes

So, how do we “reap” a child process programmatically?

Zombie?
Create & Terminate Processes

So, how do we “reap” a child process programmatically?

wait()

waitpid()
void main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
    {
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    }
    for (i = 0; i < N; i++)
    {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
int wait(int* child_status)

void main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
Create & Terminate Processes

```c
void main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

- `wait(int* child_status)`: Wait for each to terminate
- `Get Info on Status`: Get information on status
- `Create & Terminate Processes`: Create and terminate processes
void main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
Create & Terminate Processes

```c
int waitpid(-1, &status, 0)
```

is the same as...

```c
int wait(&status)
```
void main()
{
    if (fork() == 0) {
        printf("a");
    }
    else {
        printf("b");
        waitpid(-1, NULL, 0);
    }
    printf("c");
    exit(0);
}
Activity

void main()
{
    if (fork() == 0) {
        printf("a");
    }
    else {
        printf("b");
        waitpid(-1, NULL, 0);
    }
    printf("c");
    exit(0);
}

List all the possible output sequences for this program.

Solution:
We can’t make any assumption about the execution order of the parent and child. Thus, any topological sort of b -> a and a -> c is possible:

acbc
abcc
bacc
Activities in Processes

- Bursts of CPU usage alternate with periods of I/O wait
- **CPU-bound**: high CPU utilization, interrupts are processed slowly
- **I/O-bound**: more time is spending on requesting data than processing it
Process Termination

**Voluntarily**
- process finishes and asks OS to delete it (exit).

**Involuntarily**
- parent terminate execution of children processes (e.g. `TerminateProcess()` in Win32).
- Process may also terminate due to errors

**After process terminate**
- Output data from child to parent (`wait` or `waitpid`).
- Process’ resources are de-allocated by OS.

**Parent process is terminated (e.g., due to errors)**
- What will happen to the children process?!
Outline

- Basic concepts of process
  - Address space and Process control block (PCB)
- Basic operations for process management
  - Process creation/termination
- Scheduling of process: CPU scheduling
  - Basic scheduling algorithms: FIFO, SJF etc
- Inter process communication (IPC)
  - shared memory vs. message passing
Multiprogramming is a form of parallel processing in which several programs are run at the same time on a uniprocessor.

Objective?

Maximize CPU utilization. When a process wait for IO, all waiting time is wasted and no useful work is accomplished.
CPU Scheduler

- Selects one process that are ready to execute, and allocates the CPU to it (scheduler)

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (e.g., I/O request)
  2. Switches from running to ready state (e.g., quantum time passed)
  3. Switches from waiting to ready (e.g., I/O is complete)
  4. Terminates

- No choice under 1 and 4 scheduling is **nonpreemptive**

- Under 2 and 3, scheduling is **preemptive**
Scheduling Goals

- Select a process that should be executed next

All systems

- **Fairness**: give each process a fair share of the CPU
- **Balance**: keep all parts of the system busy; CPU vs. I/O
- Enforcement: ensure that the stated policy is carried out

Batch systems

- Maximize **Throughput**: the number of jobs that are completed per unit time
- Minimize **Turnaround** time: the time from submission to completion
- CPU utilization: CPU time is precious \( \rightarrow \) keep the CPU as busy as possible
Scheduling Goals

Interactive systems
- **Response/wait time**: respond quickly to users’ requests
- Proportionality: meet users’ expectations

Real-time systems: correct and in time processing
- Meet **deadlines**: deadline miss → system failure!
- Hard real-time vs. soft real-time: aviation control system vs. DVD player
- Predictability: timing behaviors is predictable
Scheduling: Which Process to Run?

- How To pick the “Lucky” Process?
- What are the scheduling objectives?
CPU Scheduling on One Core

- Execute *only one* process at a time
  - Other processes should wait for CPU

- Scheduling queues
  - **Ready queue** – processes in main memory, ready and waiting to execute
  - **Wait queue** – processes waiting for signals/interrupts
  - **Device queues** – processes waiting for I/O operations

- Processes move between the various queues
Scheduling Queues

- **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
- **Dispatcher** takes the first task from ready queue and executes it
Scheduling Queues (cont.)

- **ready queue**
- **CPU**
- **I/O**
- **I/O queue**
- **I/O request**
- **time slice expired**
- **child executes**
- **fork a child**
- **interrupt occurs**
- **wait for an interrupt**
Dispatcher

- Dispatcher gives control of CPU to the process selected by CPU scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running (overhead)
Performance Metrics for Scheduling

- **CPU utilization**
  - What percent of the time the CPU is to run programs?
  - \[ \text{util} = \frac{t_{\text{total}} - t_{\text{idle}} - t_{\text{dispatch}}}{t_{\text{total}}} \]

- **Throughput**
  - Number of processes that complete their execution per time unit

- **Turnaround time**
  - Amount of time to execute a particular process

- **Waiting time**
  - Amount of time a process has been waiting in the ready queue

- **Response time**
  - Amount of time it takes from when a request was submitted until the first response is produced
  - NOT possible to optimize for all metrics with a single scheduling algorithm
Calculate turnaround, wait, response times

- Given a process
  - Arrival time: \( t_a \)
  - First response time: \( t_r \)
  - Finish time: \( t_f \)
  - Total CPU burst time: \( t_{cpu} \)
  - Total I/O time: \( t_{io} \)

**Turnaround time**: the process spent in the system

\[ T_{turn\_arround} = t_f - t_a = t_{cpu} + t_{io} + t_{wait} \]

**Waiting time**: the process spent in the ready queue

\[ t_{wait} = (T_{turn\_arround} - t_{cpu} - t_{io}) \]

**Response time**: the process waited until the first response

\[ t_{response} = t_r - t_a \]
Classical Scheduling Algorithms

- **FCFS**: non-preemptive, based on arrival time
  - Long waiting time, e.g. long process before SSH console?

- **SJF (shortest job first)**: preemptive & non-preemptive
  - Optimal in term of waiting time

- **RR (Round-robin)**: preemptive
  - Processes take turns with fixed time quantum e.g., 10ms

- **Priority-based scheduling**
  - Real-time systems: earliest deadline first (EDF)

- **Multi-level queue (priority classes)**
  - System processes > faculty processes > student processes

- **Multi-level feedback queues**: short $\rightarrow$ long quantum
First-Come, First-Served (FCFS)

Suppose the following processes arrive at time t=0 in the given order

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

The **Gantt Chart** for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24</td>
<td>27</td>
<td>30</td>
</tr>
</tbody>
</table>

- Waiting time for $P₁ = 0$; $P₂ = 24$; $P₃ = 27$
- **Average waiting time**: $(0+24+27)/3=17$

Problem: long jobs delay every job after them. Many processes may wait for a single long job.
Suppose that the processes arrive in the order: $P_2, P_3, P_1$

- The Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>P_2</th>
<th>P_3</th>
<th>P_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$

- Average waiting time: $(6 + 0 + 3)/3 = 3$

- Convoy effect: short process behind long process
Shortest Job First (SJF)

- **Current job queue**
  - A: 4
  - B: 3
  - C: 6
  - D: 3

- **Execution order**
  - B: 3
  - D: 3
  - A: 4
  - C: 6

- **SJF scheduler**

- **Shortest job goes first**
  - Jobs are sorted in increasing order of execution time

- **Optimal for wait time**
  - Minimize average waiting time for a given set of processes
  - *Why?!*

- **Problem:** how does the scheduler know how long a job will take?
Calculate turnaround, wait, response times

Given a process
- Arrival time: $t_a$
- First response time: $t_r$
- Finish time: $t_f$
- Total CPU burst time: $t_{cpu}$
- Total I/O time: $t_{io}$

**Turnaround time:** the process spent in the system
- $T_{turn\_around} = t_f - t_a = t_{cpu} + t_{io} + t_{wait}$

**Waiting time:** the process spent in the ready queue
- $t_{wait} = (T_{turn\_around} - t_{cpu} - t_{io})$

**Response time:** the process waited until the first response
- $t_{response} = t_r - t_a$
SJF vs. SRJF-Preemptive

- A(4) and C(6) arrive at time 0
- B(2) arrives at time 1
- D(3) arrives at time 7

SJF schedule

SRJF – preemptive

\[ t_{\text{wait}} = (T_{\text{turn}} - t_{\text{cpu}} - t_{\text{io}}) \]

\[ \frac{(6-0-4)+(3-1-2)+(15-0-6)+(10-7-3)}{4} = \frac{11}{4} = 2.75 \]

Wait time: which one is better?

\[ \frac{(4-4)+(6-1-2)+(12-0-6)+(15-7-3)}{4} = \frac{14}{4} = 3.5 \]

By completion time

\[ T_{\text{wait}} = [T_{\text{f}} - (T_{\text{s}} + t_{\text{cpu}} + t_{\text{io}})] \]

\[ \frac{(6-4)+(3-3)+(15-6)+(10-10)}{4} = \frac{11}{4} = 2.75 \]
Priority Based Scheduling

- Assign a priority to each process
  - “Ready” process with highest priority allowed to run
  - Same priority: round-robin

- Priorities may be assigned dynamically
  - Reduced when a process uses CPU time
  - Increased when a process waits for I/O
Round Robin (RR) scheduling

- Round Robin scheduling
  - Each process gets a fixed time allocation (*quantum*)
  - If not complete → go back to the end of ready queue, and RR scheduler picks the next ready process in queue

- What’s a good quantum?
  - Too short: many context switches, hurt efficiency
  - Too long: poor response to interactive requests
  - Typical length: 10–50 ms
CPU Scheduling for Multiprocessors

- **Symmetric**: all CPUs run OS with self scheduling
- **Asymmetric**: one CPU runs the scheduler (master)
  - Slave CPUs: execute assigned processes
- **Process affinity**: migration or not
- **Load balancing**: push/pull process from other CPUs
- **Global queue vs. separate queues**
- **New architectures**
  - Multicore: multiple CPUs (cores) on a single machine
  - Simultaneously multithreading (SMT): multiple function units and running context (registers), more inst./cycle
CPU Scheduling for Multiprocessors (cont)

- Objectives
  - Minimal schedule length (given CPU numbers)
  - Minimal number of needed processors (other constraints)

- Priority of tasks: optimal is NP-hard
  - FIFO, or SJF
  - Longest Job First (LJF): e.g.: 7, 6, 3, 2, 2 on 2 cpus

- Scheduling algorithms
  - Worst-Fit ➔ balance workload
  - Best-Fit or First-Fit ➔ use fewer processors (for time constrained systems)
Outline

■ Basic concepts of process
  ➢ Address space and Process control block (PCB)

■ Basic operations for process management
  ➢ Process creation/termination

■ States of process: different queues
  ➢ ready, running, or wait etc

■ Scheduling of process: CPU scheduling
  ➢ Basic scheduling algorithms: FIFO, SJF etc

■ Inter process communication (IPC)
  ➢ shared memory vs. message passing (SGG 3.4, 3.5, 3.6)
Inter-Process Communication (IPC)

- Processes within a system may be **independent** or **cooperating**
- **Cooperating process** can affect or be affected by other processes
- Reasons for cooperating processes:
  - Information sharing, e.g., sharing a file
  - Computation speedup, e.g., subtasks for parallelism
  - Modularity & Convenience (e.g., editing, printing in the same time)
- Cooperating processes need **inter-process communication (IPC)**
  - Shared memory
  - Pipe and Named Pipe
  - Message passing
Inter-Process Communication (IPC)

Message Passing

Shared Memory
POSIX Shared-Memory APIs

- **shmget**: allocate a shared memory segment
  - `shm_id = shmget(IPC_PRIVATE, size, S_IRUSR|S_IWUSR);`

- **shmat**: attach the segment with one address
  - `shm_addr = (char*) shmat(shm_id, NULL, 0);`

- **shmdt**: detach the shared memory segment
  - `shmdt(shm_addr);`

- **shmctl**: alter the permission of the shared segment
  - `shmctl(shm_id, cmd, *buf);`
  - `cmd: IPC_STAT, IPC_SET, and IPC_RMID`


```c
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc, char *argv[]) {
    int shmid;  char *data;
    /* create the shared segment: */
    shmid=shmget(IPC_PRIVATE, 1024, 0644 | IPC_CREAT))
    /* attach it to a pointer */
    data=shmat(shmid, (void *)0, 0);
    /* write some data */
    sprintf(data, "Hi, I am writing share memory");
    shmdt(data); /* detach from the segment: */
    /*remove the shared segment*/
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

`ipcs -a: checking ipc information`
int main(int argc, char *argv[]) {
    int shmid; char * data;
    ... ... /* setup the shared segment: */ 
    if ( (cpid = fork())==0 ){//child process
        sprintf(data, "Child: using SM!");
        sleep(1); //give parent a chance
        printf("%s\n", data); exit(0);
    }else if (cpid >0){ //parent process
       sleep(1); //let child first
        sprintf(data, "Parent: changing SM");
        wait(cpid); //wait child to finish
    }
    shmdt(data); shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
Shared Memory, Pros and Cons

**Pros**
- Fast bidirectional communication among any number of processes
- Saves Resources

**Cons**
- Needs concurrency control (leads to data inconsistencies like ‘Lost update’)
- Lack of data protection from Operating System (OS)
Ordinary Pipes
Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style

- Producer writes to one end (the \textit{write-end} of the pipe)

- Consumer reads from the other end (the \textit{read-end} of the pipe)

- Ordinary pipes are therefore \textbf{unidirectional}

- Require parent-child/sibling relationship between communicating processes
Example of using Pipe

```c
int pfd[2];
char line[100];
main() {
  int pid;
  pipe(pfd);
  pid = fork();
  if (pid == 0) {
    ChildFunc();
  } else {
    ParentFunc();
    waitpid(NULL);
  }
}

Void ParentFunc() {
  close(pfd[1]);
  read (pfd[0], line, sizeof(line));
  printf("date from child is: %s\n", line);
  close(pfd[0]);
}

Void ChildFunc() {
  close(pfd[0]);
  dup2(pfd[1], 1);
  close(pfd[1]);
  execl("/bin/date", "date", 0);
}

Demo: pipe.c
```
Problems of Pipe

■ Pros
  ➢ simple
  ➢ flexible
  ➢ efficient communication

■ Cons:
  ➢ no way to open an already existing pipe. This makes it impossible for two arbitrary processes to share the same pipe, unless the pipe was created by a common ancestor process.
Named Pipes (FIFO)

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

- Named pipes also allow two unrelated processes to communicate with each other.
- They are also known as FIFOs (first-in, first-out)
- Used to establish a one-way (half-duplex) flow of data.
**Named Pipes Cont.**

- **How does it work?**
  - Uses an access point (A file on the file system)
  - Two unrelated processes opens this file to communicate
  - One process writes and the other reads, thus it is a half-duplex communication
Named Pipes - Properties

- Named pipes are system-persistent objects.
  - i.e. They exist beyond the life of the process
  - In contrary, anonymous pipes are process-persistent objects

- Named Pipes must be explicitly deleted by one of the process by calling “unlink”

- By default, it supports blocked read and writes.
  - i.e. if a process opens the file for reading, it is blocked until another process opens the file for writing, and vice versa.
  - This can be overridden by calling O_NONBLOCK flag when opening them.

- A named pipe must be opened either read-only or write-only because it is half-duplex, that is, a one-way channel.
Code Example of Half-Duplex Communication
How to obtain Full-Duplex using Named Pipes?

- A full-duplex communication can be established using different named pipes so each named pipe provides the flow of data in one direction.

- Care should be taken to avoid deadlock.
Code Example of Full-Duplex Communication

Client

Pipe NP1

Pipe NP2

Server
Message Passing

- A message-passing facility provides at least:
  - `send(message)`
  - `receive(message)`
  - Communication: direct or indirect
  - Operations: blocking (synchronous) or non-blocking (asynchronous)

- General communication mechanism
  - Processes running on different computers

**When to use shared memory vs. message passing?**
Message Passing in Client-Server Systems

Low-level communication
- Sockets - end-point for comm. (ip address + port number)
- Server creates a socket and accepts requests from clients.
- Client makes connection request to the server socket, and read/write messages to the socket.

High-level communication
- RPC
  - Invoke a procedure on a remote host
  - Hide communication details
- RMI
  - Java feature similar to RPC
  - Invoke a method on a remote object
IPC with Message Passing (socket)

C/C++ (sys/socket.h, netinet/in.h)

- **Server**
  - Create a socket with the `socket()`
  - Bind the socket to an address using the `bind()`
  - Listen for connections with the `listen()`
  - Accept a connection with the `accept()` system call.

- **Client**
  - Create a socket with the `socket()` system call
  - Connect to server using the `connect()` system call
  - `read()` and `write()`

Java

- **Server:** `ServerSocket`
- **Client:** `Socket`
Summary

- Basic concepts of process
  - Representation: process control block (PCB)
  - Execution environment: address space

- Basic operations for process management
  - Process creation/termination

- States of process: different queues
  - ready, running, or wait etc
  - Context switch: multiple hardware running contexts

- Scheduling of process: CPU scheduling
  - Basic scheduling algorithms: FIFO, SJF etc

- Inter process communication