FAQ

• Why not let the parent process do everything? Because ...

• Where does the child process begin execution?
  – From fork (). It returns the value 0 in the child process.
  – In the parent fork () returns the PID of the child.

• fork (): does parent run before child? parent already running.

• Questions on wait() example: rv = wait(&wstatus);
  – Caller will block until the child exists.
  – on success, returns PID of the terminated child; on error, -1 is returned.
  – Status in wstatus variable, extracted using WEXITSTATUS(wstatus)
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t cid;

    /* fork a child process */
    cid = fork();
    if (cid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed\n");
        return 1;
    }
    else if (cid == 0) { /* child process */
        printf("I am the child %d, my PID is %d\n", cid, getpid());
        execvp="/bin/ls","ls",NULL);
    } 
    else { /* parent process */
        /* parent will wait for the child to complete */
        printf("I am the parent with PID %d, my parent is %d, my child is %d\n",getpid(), getppid(), cid);
        wait(NULL);

        printf("Child Complete\n");
    }

    return 0;
}
FAQ

• Where does the child process begin execution?
  – From fork ( ). It returns the value 0 in the child process.
  – In the parent fork ( ) returns the PID of the child.
  – After fork ( ) does parent run before the child? parent already running.

• Questions on wait( ) example: \( rv = \text{wait}(\&\text{wstatus}); \)
  – Caller will block until the child exists.
  – on success, returns the PID of the terminated child; on error, -1 is returned.
  – Status in wstatus variable, extracted using WEXITSTATUS(wstatus)
FAQ: Buffering

- Shared data
  ```c
  #define BUFFER_SIZE 8
  typedef struct {
      ...
  } item;

  item buffer[BUFFER_SIZE];
  int in = 0;
  int out = 0;
  ```

Why do we need buffers?
- The producer and the consumer process operate at their own speeds. Items wait in the buffer when consumer is slow.

Where does the bounded buffer “start”?
- It is circular
Interprocess Communication – Shared Memory

• An area of memory shared among the processes that wish to communicate
• The communication is under the control of the users processes, not the operating system.
• Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
  – Synchronization is discussed in great details in Chapter 6.
• Example soon.
Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - `send(message)`
  - `receive(message)`
- The `message` size is either fixed or variable
• If processes $P$ and $Q$ wish to communicate, they need to:
  – Establish a *communication link* between them
  – Exchange messages via send/receive

• Implementation issues:
  – How are links established?
  – Can a link be associated with more than two processes?
  – How many links can there be between every pair of communicating processes?
  – What is the capacity of a link?
  – Is the size of a message that the link can accommodate fixed or variable?
  – Is a link unidirectional or bi-directional?
• Implementation of communication link
  – Physical:
    • Shared memory
    • Hardware bus
    • Network
  – Logical: Options (details next)
    • Direct (process to process) or indirect (mail box)
    • Synchronous (blocking) or asynchronous (non-blocking)
    • Automatic or explicit buffering
Direct Communication

• 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 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)
  – Each mailbox has a unique id
  – Processes can communicate only if they share a mailbox

• Properties of communication link
  – Link 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
  – Link may be unidirectional or bi-directional
Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:
  - \texttt{send}(A, \textit{message}) – send a message to mailbox A
  - \texttt{receive}(A, \textit{message}) – receive a message from mailbox A
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?

• Possible 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. Sender is notified who the receiver was.
Synchronization (blocking or not)

• Message passing may be either blocking or non-blocking

• **Blocking** is termed *synchronous*
  – **Blocking send** -- sender is blocked until message is received
  – **Blocking receive** -- receiver is blocked until a message is available

• **Non-blocking** is termed *asynchronous*
  – **Non-blocking send** -- sender sends message and continues
  – **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message

☐ Different combinations possible
  ☐ If both send and receive are blocking, we have a **rendezvous**.
  ☐ Producer-Consumer problem: Easy if both block
Examples of IPC Systems - POSIX

- Older scheme (System V) using shmget(), shmat(), shmdt(), shmctl()

- POSIX Shared Memory
  - Process first creates shared memory segment
    ```c
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ```
    - Returns file descriptor (int) which identifies the file
  - Also used to open an existing segment to share it
  - Set the size of the object
    ```c
    ftruncate(shm_fd, 4096);
    ```
  - map the shared memory segment in the address space of the process
    ```c
    ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    ```
  - Now the process could write to the shared memory
    ```c
    sprintf(ptr, "Writing to shared memory");
    ```
Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Other process opens shared memory object `name`
    ```c
    shm_fd = shm_open(name, O_RDONLY, 0666);
    ```
  - Returns file descriptor (int) which identifies the file
  - map the shared memory object
    ```c
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    ```
  - Now the process can read from to the shared memory object
    ```c
    printf("%s", (char *)ptr);
    ```
  - remove the shared memory object
    ```c
    shm_unlink(name);
    ```
#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 object */
    char* 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", message1);
    ptr += strlen(message_1);
    return 0;
/* create the shared memory segment */
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

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

/* now map the shared memory segment in the address space of the process */
ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
if (ptr == MAP_FAILED) {
    printf("Map failed\n");
    return -1;
}

/**
  * Now write to the shared memory region.
  *
  * Note we must increment the value of ptr after each write.
  */
sprintf(ptr,"%s",message0);
ptr += strlen(message0);
sprintf(ptr,"%s",message1);
ptr += strlen(message1);
sprintf(ptr,"%s",message2);
ptr += strlen(message2);

return 0;
#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 object */
    char *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;
}
/* open the shared memory segment */
shm_fd = shm_open(name, O_RDONLY, 0666);
if (shm_fd == -1) {
    printf("shared memory failed\n");
    exit(-1);
}

/* now map the shared memory segment in the address space of the process */
ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
if (ptr == MAP_FAILED) {
    printf("Map failed\n");
    exit(-1);
}

/* now read and print from the shared memory region */
printf("%s",ptr);

/* remove the shared memory segment */
if (shm_unlink(name) == -1) {
    printf("Error removing %s\n",name);
    exit(-1);
}
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)
Socket Communication

- CS457 Computer Networks and the Internet
Conduit allowing two processes to communicate

- **Ordinary ("anonymous") pipes** – Typically, a parent process creates a pipe and uses it to communicate with a child process that it created. Cannot be accessed from outside the process that created it. Created using `pipe()` in Linux.

- **Named pipes ("FIFO")** – can be accessed without a parent-child relationship. Created using `fifo()` in Linux.
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** (half duplex)

**Require parent-child relationship** between communicating processes

- `pipe (int fd[])` to create pipe, `fd[0]` is the read-end, `fd[1]` is the write-end

Windows calls these **anonymous pipes**

For a process the **file descriptors** identify specific files.
Ordinary Pipes

- Pipe is a special type of file.
- Inherited by the child
- Must close unused portions of the pipe
UNIX pipe example \( \frac{1}{2} \) (parent)

```c
#define READ_END  0
#define WRITE_END 1

int fd[2];

create the pipe:
    if (pipe(fd) == -1) {
        fprintf(stderr,"Pipe failed");
        return 1;
    }

fork a child process:
    pid = fork();

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]);
```

---

**Direction of flow**

- **Parent Process:**
  - Close the unused end of the pipe.
  - Write to the pipe.
  - Close the write end of the pipe.

- **Child Process:**
  - Inherits the pipe.
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("child read %s\n", read_msg);

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

- Named Pipes (termed FIFO) 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
Chapter 4: Threads

Objectives:

• Thread—basis of multithreaded systems
• APIs for the Pthreads and Java thread libraries
• implicit threading, multithreaded programming
• OS support for threads
Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples
Modern applications are multithreaded

• Most modern applications are multithreaded
  – Became common with GUI
• 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
Multithreaded Server Architecture

(1) request

(2) create new thread to service the request

(3) resume listening for additional client requests
Benefits

• **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 (10-100 times), thread switching lower overhead than context switching

• **Scalability** – process can take advantage of multiprocessor architectures
Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- **Parallelism** implies a system can perform more than one task simultaneously
  - Extra hardware needed for parallel execution
- **Concurrency** supports more than one task making progress
  - Single processor / core: scheduler providing concurrency
Concurrent execution on single-core system:

Parallelism on a multi-core system:
Types of parallelism

- **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
- **Task parallelism** – distributing threads across cores, each thread performing unique operation

As # of threads grows, so does architectural support for threading

- CPUs have cores as well as *hardware threads*
  - *e.g. hyper-threading*
    - Oracle SPARC T4 with 8 cores, and 8 hardware threads per core (total 64 threads)
    - AMD Ryzen 7 with 4 cores and 8 threads
Single and Multithreaded Processes

![Diagram showing single-threaded process and multithreaded process]

- **Single-threaded process**
  - Code
  - Data
  - Files
  - Registers
  - Stack
  - Thread

- **Multithreaded process**
  - Code
  - Data
  - Files
  - Registers
  - Registers
  - Registers
  - Stack
  - Stack
  - Stack
  - Thread
Process vs Thread

• All threads in a process have same address space (text, data, open files, signals etc.), same global variables

• Each thread has its own
  – Thread ID
  – Program counter
  – Registers
  – Stack: execution trail, local variables
  – State (running, ready, blocked, terminated)

• Thread is also a schedulable entity
Amdahl’s Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components.
- $S$ is serial portion (as a fraction).
- $N$ processing cores.

\[
\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}
\]

- **Example**: 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
  – Mac OS X
Multithreading Models

How do kernel threads support user process threads?

- Many-to-One
- One-to-One (now common)
- Many-to-Many
Many-to-One

- Many user-level threads mapped to single kernel thread (thread library in user space)
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads for Java 1996
  - GNU Portable Threads 2006
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later
Many-to-Many Model

- Allows many user level threads to be mapped to smaller or equal number of kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9 (2002-3)
- Windows with the *ThreadFiber* package (NT/2000)
Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX -2006
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier
Single and Multithreaded Processes

![Diagram showing single-threaded and multithreaded processes](image-url)
Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS