CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Fall 2022 Lecture 6
Processes

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
FAQ

Programs with multiple processes is a new paradigm for you!

- Why are child processes needed? Can they have their own child processes?

- When does the child process begin execution? fork()

- What does fork() return?
  - It returns the value 0 in the child process. Child’s PID is not zero
  - In the parent fork() returns the PID of the child.

- What do they return?: getpid(), getppid()

- Fork is not a branch or a function call like the ordinary programs you have worked with in the past. The child process is a separate process.

- Fork is the only way to create a process (after init).
FAQ

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

• If the child has exited and the parent hasn’t yet executed \texttt{wait( )}.
  – The child is in terminated (zombie) state.

• Self exercise 2: Examine, compile and run programs.
Electronic devices in lecture room

• Use of Laptops, phones and other devices are not permitted.
• Exception: only with the required **pledge** that you will
  – Must have a reason for request
  – use it only for class related note taking, which **must be submitted on 1st and 15th** of each month.
  – not distract others, turn off wireless, last row
• [Laptop use lowers student grades, experiment shows, Screens also distract laptop-free classmates](#)
• [The Case for Banning Laptops in the Classroom](#)
• [Laptop multitasking hinders classroom learning for both users and nearby peers](#)

Permitted students: Ray B., Heidi L., Sawyer P.
Classroom Change?

• iClicker Cloud issues: under investigation
• Is it the room, or the iCloud server?
• Guest network instead of the EID network?
#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());
        execlp("/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;
}

Parent and the child processes run concurrently.
Interprocess 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 **interprocess communication (IPC)**
• Two models of IPC
  – Shared memory
  – Message passing
Producer-Consumer Problem

• Common paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  – **unbounded-buffer** places no practical limit on the size of the buffer
  – **bounded-buffer** assumes that there is a fixed buffer size

Why do we need a buffer (shared memory region)?
- 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
Bounded-Buffer – Shared-Memory Solution

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

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

- **in** points to the **next free position** in the buffer.
- **out** points to the **first full position** in the buffer.
- Buffer is empty when **in == out**;
- Buffer is full when **((in + 1) % BUFFER_SIZE) == out**. (Circular buffer)
- This scheme can only use **BUFFER_SIZE-1** elements

![Circular Buffer Diagram](image)
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* consume the item in next consumed */
}
• Each process has its own private address space.
• An area of memory shared among the processes that wish to communicate
• The communication is under the control of the user processes, not the operating system.
• Major issue 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 a later Chapter.
• POSIX 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
Message Passing (Cont.)

• 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 *(mailbox)*
    • Synchronous *(blocking)* or asynchronous *(non-blocking)*
    • Automatic or explicit buffering
Direct Communication

• Processes must name each other explicitly:
  – send \((P, \text{message})\) – send a message to process \(P\)
  – receive \((Q, \text{message})\) – 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.
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

OSs support many different forms of IPC*. We will look at two of them

- Shared Memory
- Pipes

* Linux kernel supports: Signals, Anonymous Pipes, Named Pipes or FIFOs, SysV Message Queues, POSIX Message Queues, SysV Shared memory, POSIX Shared memory, SysV semaphores, POSIX semaphores, FUTEX locks, File-backed and anonymous shared memory using mmap, UNIX Domain Sockets, Netlink Sockets, Network Sockets, Inotify mechanisms, FUSE subsystem, D-Bus subsystem
Older scheme (System V) used `shmget()`, `shmat()`, `shmdt()`, `shmctl()`

**POSIX Shared Memory**

- First process first creates shared memory segment
  ```c
  shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
  ```
  - Returns file descriptor (int)
  - Identified by name (string)
  - 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");
  ```
Ex. POSIX Shared memory (2)

- **POSIX Shared Memory**
  - Other process opens shared memory object \texttt{name}
    \[
    \texttt{shm\_fd = shm\_open(name, O\_RDONLY, 0666)};
    \]
    - Returns file descriptor (int) which identifies the file
  - map the shared memory object
    \[
    \texttt{ptr = mmap(0,SIZE, PROT\_READ, MAP\_SHARED,}
    \]
    \[
    \texttt{\quad shm\_fd, 0)};
    \]
    - Now the process can read from to the shared memory object
  - printf("%s", (char *)\texttt{ptr});
  - remove the shared memory object
    \[
    \texttt{shm\_unlink(name)};
    \]

Please remember to unlink, name persists in OS.
#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;

    /* 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;
```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 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
- Pipes
- Remote Procedure Calls
  - Calling a function on another machine through the network.
- Remote Method Invocation (Java)
  - Object oriented version of RPC
Socket Communication

- CS457 Computer Networks and the Internet

80: HTTP (well known)
Pipes

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.
  - Ends identified by file descriptors (FDs).
- Inherited by the child
- Flow: from Write End of P/C to Read End of C/P
  - Must close unused portions of the pipe
- Next example: Parent to child information flow
UNIX pipe example 1/2 (parent)

#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]);

Child 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
CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Threads

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
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

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

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

Gives speedup from adding additional cores to an application that has both serial and parallel components.

- $S$ is serial portion (as a fraction) that cannot be broken into parallel operations.
- Some things can possibly be done in parallel.
- $N$ 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 
\[
1/(0.25+ 0.75/2) = 1.6 \text{ times}
\]

- As $N$ approaches infinity, speedup approaches $1/S$

Serial portion of an application has disproportionate effect on performance gained by adding additional cores
Amdahls law: ordinary life example.

Which of the two option is faster?

- Person A cooks, person B eats and then Person C eats.
- Person A cooks, then both person B and person C eat at the same time.
User Threads and Kernel Threads

• **User threads** - management done by user-level threads library
  
  Three main thread libraries:
  – POSIX Pthreads
  – Windows threads
  – Java threads

• **Kernel threads** - Supported by the Kernel
  – Examples – virtually all general-purpose operating systems, including:
  • Windows
  • Linux
  • Mac OS X
Multithreading Models

How do kernel threads support user process threads?

• Many-to-One: Many user-level threads mapped to single kernel thread *(thread library in user space older model)*

• One-to-One: *(now common)*

• Many-to-Many: Allows many user level threads to be mapped to smaller or equal number of kernel threads *(older systems)*