CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2018 Lecture 6
Processes

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

• If there is no buffer (buffer size =0)?
  • Exact timing or they wait for each other.

• Init adapts an orphan process

• What does wait( ) return? `pid = wait(&status);`
  - on success, returns the process ID of the terminated child; on error, -1 is returned.
FAQ: Buffering

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

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

  ```
  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;
      }
  }
  ```
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
Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
  1. Zero capacity – no messages are queued on a link. Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of $n$ messages Sender must wait if queue full
  3. Unbounded capacity – infinite length Sender never waits
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");
    ```
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 = "DS";
    /* 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 */
    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 */
    printf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    printf(ptr, "%s", message_1);
    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 */
fruncate(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;
IPC POSIX Consumer

```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 */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, 0_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 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);
    }
Output generated by the programming assignment

Producer: Placed 37 at Location 0 at Time: 2017-11-10 16:06:59.00916
Consumer 2: Removed 37 from location: 0 at Time: 2017-11-10 16:06:59.00917
Consumer 1: Unable to consume, buffer empty, at Time: 2017-11-10 16:06:59.00918
Producer: Placed 5 at Location 1 at Time: 2017-11-10 16:06:59.00918
Consumer 2: Removed 5 from location: 1 at Time: 2017-11-10 16:06:59.00919
Producer: Placed 70 at Location 2 at Time: 2017-11-10 16:06:59.00919
Consumer 2: Removed 70 from location: 2 at Time: 2017-11-10 16:06:59.00919
Communications in Client-Server Systems

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

• A **socket** is defined as an endpoint for communication

• Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host

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

• Communication consists between a pair of sockets

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

• Special IP address **127.0.0.1** (**loopback**) to refer to system on which process is running
Socket Communication

- CS457 Computer Networks and the Internet
Pipes

• Acts as a conduit allowing two processes to communicate
• One of the first IPC implementation mechanisms
Pipes

• Conduit allowing two processes to communicate

• Issues:
  – Is communication unidirectional or bidirectional?
  – If bidirectional, is it **half-duplex** (one way at a time) or **full-duplex** (both directions simultaneously)?
  – Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
  – Can the pipes be used over a network?
Pipes

• Command line:
  – Set up pipe between commands
    `ls | more`
  Output of `ls` delivered as input to more

• 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.

• Named pipes ("FIFO") – can be accessed without a parent-child relationship.
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**
Ordinary Pipes

- Pipe is a special type of file.
- Inherited by the child
- Must close unused portions of the pipe
UNIX pipe example 1/2

```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]);
```

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
Fall 2018  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
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

[Diagram showing a client sending a request to a server, which creates a new thread to service the request, and then resumes listening for additional 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
Multicore Programming (Cont.)

• 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*
  – Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core
Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**
  
  ![Single core concurrency diagram]

- **Parallelism on a multi-core system:**
  
  ![Multi-core parallelism diagram]
Single and Multithreaded Processes

- Single-threaded process
- Multithreaded process
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 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
• How many partners can we cave for project:
  – Research: 3-4, Raspberry Pi: 2
• Parallel vs serial
• Amdahl's law:
  – 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.
• Thread management: A user thread can be managed by a user-level library (not managed by OS) or be managed by OS.
Questions from last time

• How threads and processes work together?
  – Process has one of more threads
• What are pipes? Functions, arrays, strings?
  – Special kind of files
• Benefits of having pipes
• Pipes vs other IPC methods
• Pipes vs message passing?
• Questions of threads:
  – Multithreading
  – Implicit threading
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

![Diagram showing the relationship between user threads and kernel threads for a two-level Model](image-url)
Single and Multithreaded Processes

- Single-threaded process
- Multithreaded process

- Code
- Data
- Files
- Registers
- Stack
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