Operating System

Yuqiao Meng

2021–9–124

Contents

1 Overview

1.1 What?

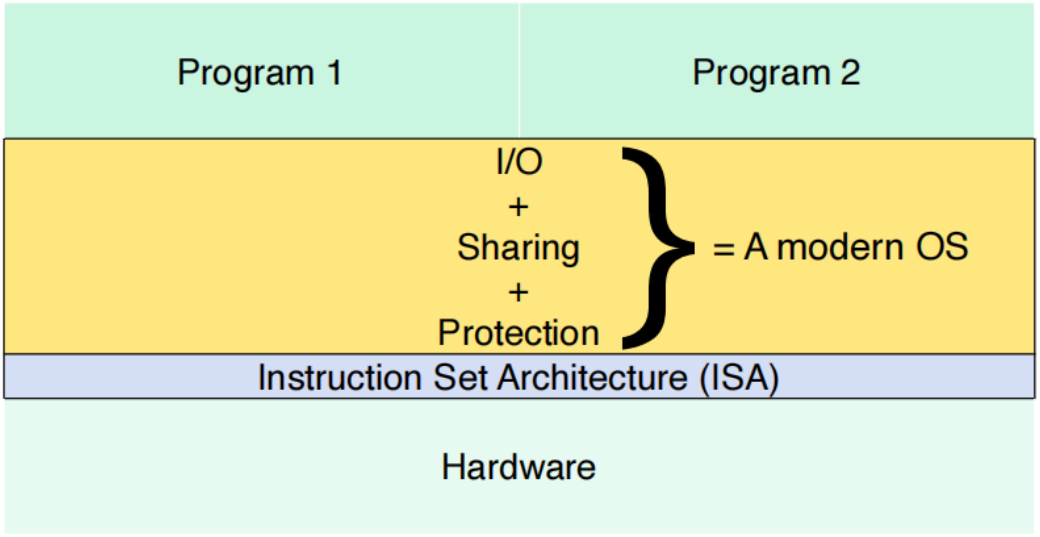
What is an Operating System? What's its reponsibility?

- A bunch of software and data residing somewhere in memory.
- The most privileged software in a computer. It can do special things, like write to disk, talk over the network, control memory and CPU usage, etc
- Manages all system resources, including CPU, Memory, and I/O devices.

1.2 Why?

Why do we need an OS?

- OS helps program to control hardwares.
- OS determines the way programs share resources.
- OS protects hardwares and programs from getting attacked.
- OS stores files persistently.



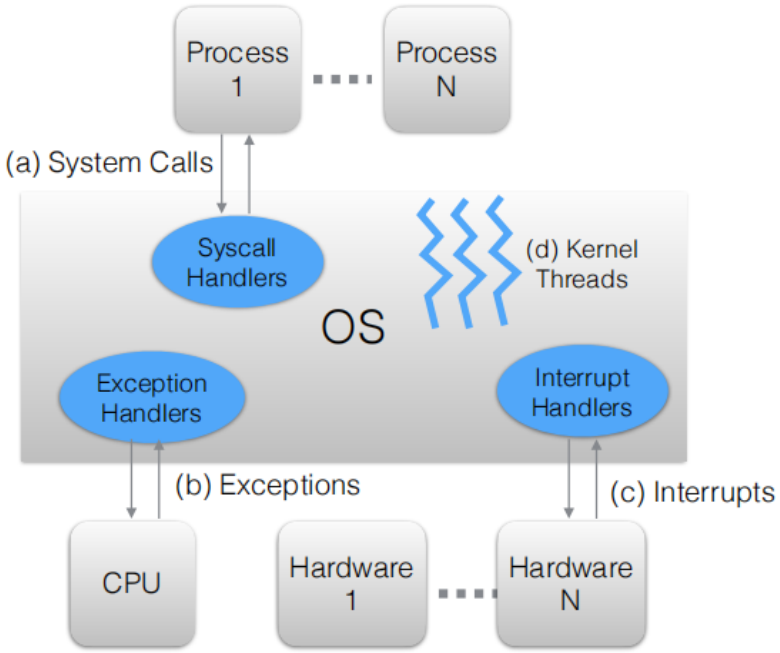
1.3 How

1.3.1 Virtulization

- Definition: OS takes a physical resource (such a sthe processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use virtual form of itself.
- Resource Virtulization
  - Many(virtual)-to-one(physical): CPU Virtulization
  - One-to-many: Disk Virtulization
  - Many-to-many

1.3.2 How to invoke OS code?

- System calls: Function calls into the OS, that OS provides these calls to run programs, access memory and devices, and other related actions.
- Exceptions: CPU will raise an exception to the OS when the running program do something wrong
- Interrupts: Hardwares send interrupts to invoke OS
- Kernel Threads: Programs run in the kernel context, executing kernel level functions.

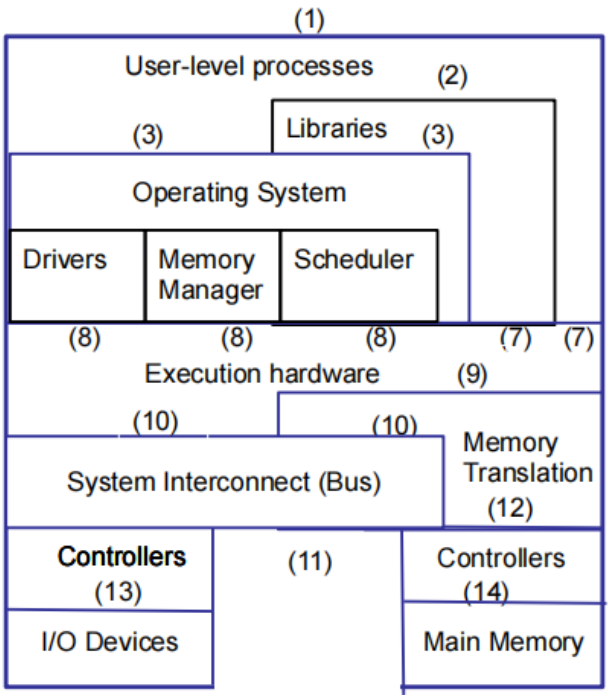


1.4 Interface

1.4.1 Explaination

- Instruction Set Architecture(ISA): the language CPU understand
- User ISA: ISA that any program can execute, it's accessible for all programs, doesn't need the service of operating system
- System ISA: ISA that only operating system is allowed to execute.
- Application Binary Interface(ABI): the combination of syscalls and User ISA(3, 7), it's the view of the world, seen by programs.
- Application Programmers' Interface(API): the combination of libraries and User ISA(2, 7), it's the tools programmer use to write codes.

1.4.2 Interfaces in a Computer System



- User ISA: 7
- System ISA: 8
- Syscalls: 3
- Application Binary Interface: 3, 7
- Application Programmers' Interface: 2, 7

1.5 History

- First Computer: Atanasoff–Berry computer, or ABC.
- First OS: GM-NAA I/O, produced in 1956 by General Motors' Research division for its IBM 704.
- First language: Plankalkül, developed by Konrad Zuse for the Z3 between 1943 and 1945.

- First programmer: Ada Lovelace

2 Processes

2.1 Process

2.1.1 What?

What is a process?

- A process is a program in execution. A program is a set of instructions somewhere (like the disk).
- Once created, a process continuously does the following:
  - **Fetches** an instruction from memory.
  - **Decodes** it. i.e., figures out which instruction this is.
  - **Executes** it. it does the thing that it is supposed to do, like add two numbers together, access memory, check a condition, jump to a function, and so forth.

2.1.2 Process versus Program

How is a process different from a program?

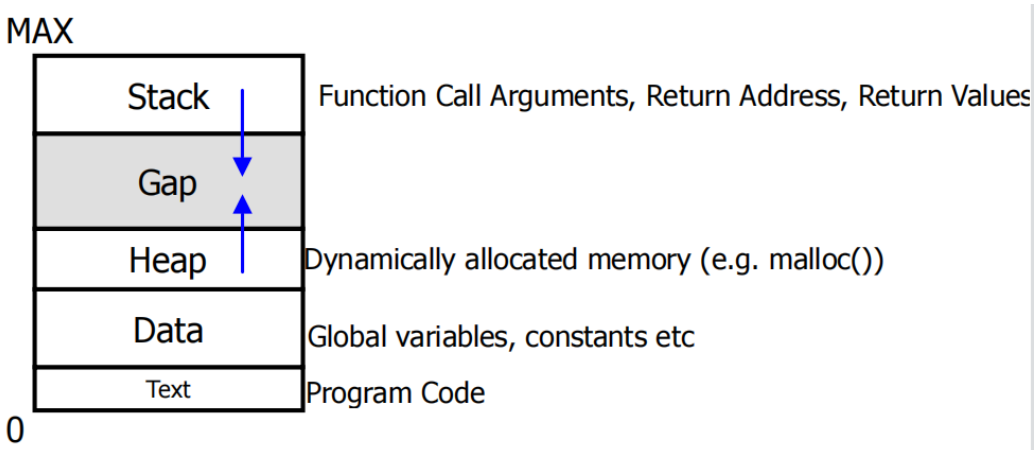
- Program: A passive entity stored in the disk, has static code and static data.
- Process: Actively executing code and the associated static and dynamic data.
- Program is just one component of a process.
- There can be multiple process instances of the same program

2.1.3 Constitution

- Memory space
- Procedure call stack

- Registers and counters
- Open files, connections
- And more.

2.1.4 Memory layout



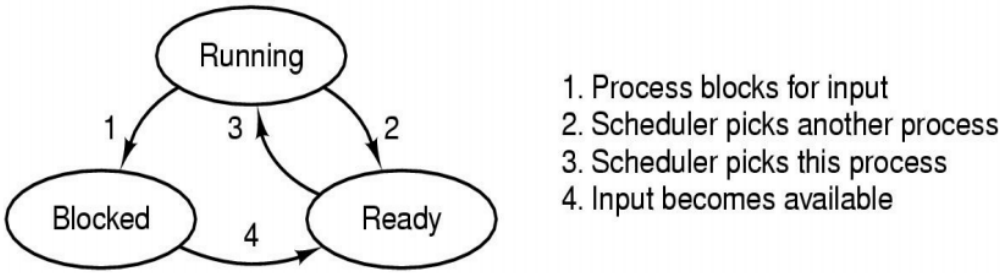
In this picture, Stack and Heap grow toward each other, that's because every process has a limited amount of space, thus let heap and stack grow toward each other from two direction can make the best use of space.

2.2 System calls

- fork(): create new process. **called once but return twice**. Usage:
  - User runs a program at command line
  - OS creates a process to provide a service: Check the directory /etc/init.d/ on Linux for scripts that start off different services at boot time.
  - One process starts another process: For example in servers
- exec(): execute a file. **replaces the process' memory with a new program image. All I/O descriptors open before exec stay open after exec**.
- wait()/waitpid(): wait for child process.

- exit(): terminate a process

2.3 Lifecycle



- Ready (runnable; temporarily stopped to let another process run)
  - Process is ready to execute, but not yet executing
  - Its waiting in the scheduling queue for the CPU scheduler to pick it up.
- Running: (actually using the CPU at that instant)
- Blocked (unable to run until some external event happens).
  - Process is waiting (sleeping) for some event to occur.
  - Once the event occurs, process will be woken up, and placed on the scheduling queue

1. Running → Blocked: Occurs when the operating system discovers that a process cannot continue right now.
2. Running → Ready: Occurs when the scheduler decides that the running process has run long enough and it is time to let another process have some CPU time.
3. Ready → Running: Occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again.
4. Blocked → Ready: Occurs when the external event for which a process was waiting (such as the arrival of some input) happens

2.4 Special Process

- Orphan process
  - When a parent process dies, child process becomes an orphan process
  - The init process (pid = 1) becomes the parent of the orphan processes
- Zombie process
  - When a child dies, a SIGCHLD signal is sent to the OS, If parent doesn't wait()on the child, and child exit()s, it becomes a zombie.
  - Zombies hang around till parent calls wait() or waitpid().
  - Zombies take up no system resources, it's just a integer status kept in the OS.
  - Ways to prevent a child process from becoming a zombie:
    * Parent call wait()/waitpid() before child process exit()
    * Child parent sleep() before exit() until parent process give it a message.
    * Set act.sa_flags is SA_NOCLDWAIT

2.5 Cold-start Penalty

2.6 Context Switch

3 Inter-Process Communication

3.1 Overview

Inter-Process Communication mechanisms

- Pipe:
- Signals: Event notification from one process to another
- Shared momery: Common piece of read/write memory, needs authorization for access

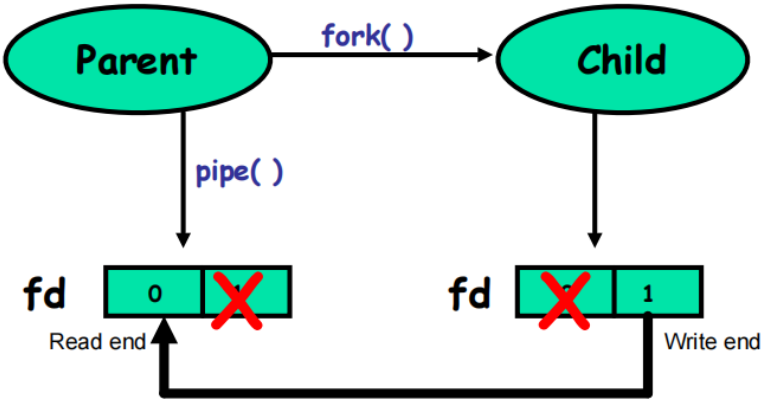
- Parent-child: Command-line arguments, including waitpid(), wait(), exit()
- Reading/modifying common files
- Semaphores: Locking and event signaling mechanism between processes
- Sockets: Not just across the network, but also between processes

3.2 Pipe

3.2.1 Abstraction

Write to one end, read from another.

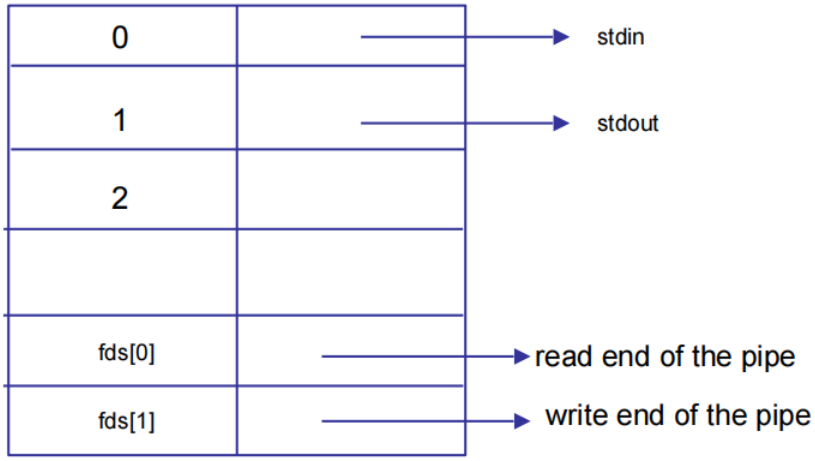
3.2.2 Parent-Child Communication Using Pipe



3.2.3 File-Descriptor Table

- Each process has a file-descriptor table
- One entry for each open file

- File includes regular file, stdin, stdout, pipes, etc.



3.2.4 Redirect Std to a File

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fds[2];
    char buf[30];
    pid_t pid1, pid2, pid;
    int status, i;

    /* create a pipe */
    if (pipe(fds) == -1) {
        perror("pipe");
        exit(1);
    }

    /* fork first child */
    if ( (pid1 = fork()) < 0) {
```

```
    perror("fork");
    exit(1);
 }

 if ( pid1 == 0 ) {
    close(1); /* close normal stdout (fd = 1) */
    dup2(fds[1], 1); /* make stdout same as fds[1] */
    close(fds[0]); /* we don't need the read end -- fds[0] */

    if( execlp("ps", "ps", "-elf", (char *) 0) < 0) {
        perror("Child");
        exit(0);
    }

/* control never reaches here */
 }

 /* fork second child */
 if ( (pid2 = fork()) < 0) {
perror("fork");
exit(1);
 }

 if ( pid2 == 0 ) {
    close(0);  /* close normal stdin (fd = 0)*/
    dup2(fds[0],0); /* make stdin same as fds[0] */
    close(fds[1]); /* we don't need the write end -- fds[1]*/

    if( execlp("less", "less", (char *) 0) < 0) {
        perror("Child");
        exit(0);
    }

/* control never reaches here */
 }

 /* parent doesn't need fds - MUST close - WHY? */
 /* The reading side is supposed to learn that the writer has
    finished if it notices an EOF condition. This can only
    happen if all writing sides are closed.*/
```

```
    /* close its reading end (for not wasting FDs and for proper
       detection of dying reader)*/
    /* close its writing end (in order to be possible to detect the
       EOF condition).*/
    close(fds[0]);
    close(fds[1]);

    /* parent waits for children to complete */
    for( i=0; i<2; i++) {
   pid = wait(&status);
   printf("Parent: Child %d completed with status %d\n", pid,
      status);
    }

}
```

3.2.5 Handling Chain of Filters Using Pipe

command 1 | *command 2* | . . . | *command N*

- First command?
  - Yes: continue
  - No: redirect stdin to previous pipe
- Last command?
  - Yes: Output
  - No:
    - create next pipe (if needed)
    - redirect stdout to next pipe
    - fork a child for next level of recursion with one command less as input
- exec the command for the current level

3.2.6 Byte-stream versus Message

- Byte-Stream abstraction: Pipe
  - Can read and write at arbitrary byte boundaries
  - Don't need to return explicit bytes of data. *read(fds[0], buf, 6)*
    - read() could reach end of input stream (EOF).
    - Other endpoint may abruptly close the connection
    - read() could return on a signal
- Message abstraction: Provides explicit message boundaries.

3.2.7 Error Handling

We must incorporate error handling with every I/O call (actually with any system call)

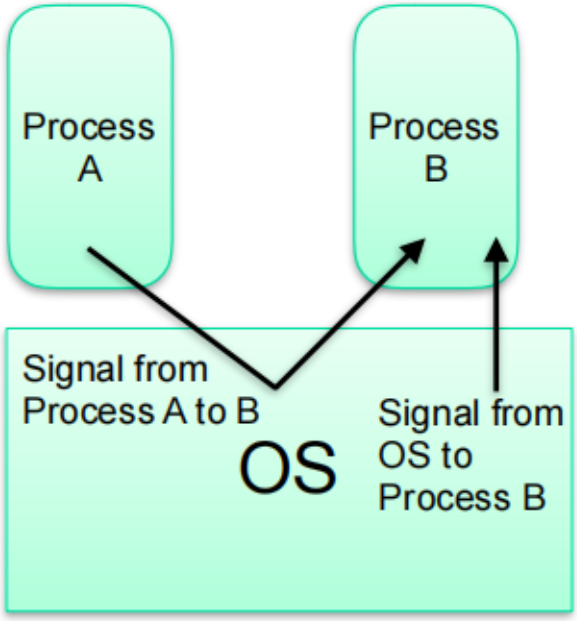
- First check the return value of every read(...)/write(...) system call
- Then
  - Wait to read/write more data
  - Handle any error conditions

```
More convinient to write a wrapper function
/* Write "n" bytes to a descriptor. */
ssize_t  writen(int fd, const void *vptr, size_t n)
{
      size_t        nleft;
      size_t       nwritten;
      const char     *ptr;

      ptr = vptr;
      nleft = n;
      while (nleft > 0) {
          if ((nwritten = write(fd, ptr, nleft))<=0){
              if (errno == EINTR)
                 nwritten = 0;  /* call write() again*/
              else return(-1);  /* error */
           }
           nleft -= nwritten;
           ptr   += nwritten;
      }
      return(n);

}
```

3.3 Signals

3.3.1 Overview

- A notification to a process that an event has occurred, comes from OS or another process
- The type of event determined by the type of signal



3.3.2 Handling Signals

- Signals can be **caught** – i.e. an action (or handler) can be associated with them
- Actions can be customized using /emphsigaction(), which associates a signal handler with the signal
- **Default** action for most signals is to terminate the process. Except SIGCHLD and SIGURG are ignored by default
- Unwanted signals can be **ignored**, except SIGKILL or SIGSTOP

3.3.3 SIGCHLD

- Sent to parent when a child process terminates or stops
- If act.sa_handler is SIG_IGN, SIGCHLD will be ignored (default behavior)
- If act.sa_flags is SA_NOCLDSTOP, SIGCHLD won't be generated when children stop

- If act.sa_flags is SA_NOCLDWAIT, children of the calling process will not be transformed into zombies when they terminate
- These need to be set in sigaction() before parent calls fork()

Usage: handling child's exit without blocking on wait()

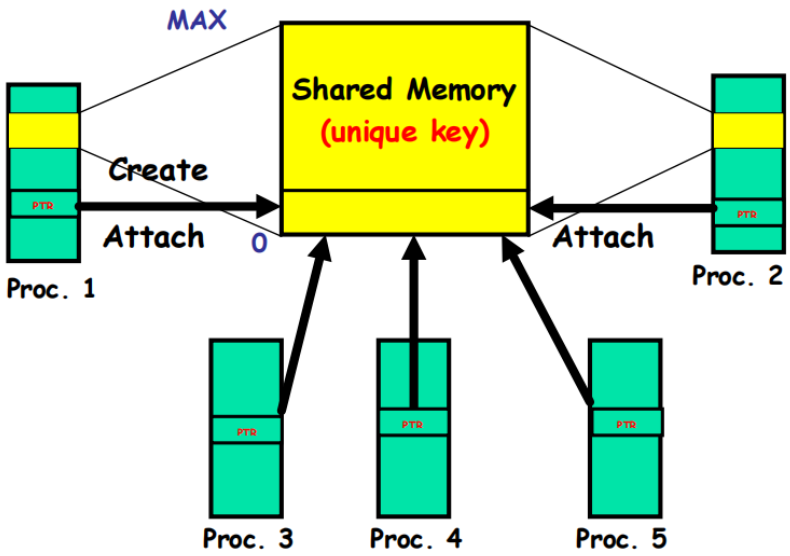
- Parent could install a signal handler for SIGCHLD
- Call wait(...)/waitpid(...)inside the signal handler

```
// sigchild.c
void handle_sigchld(int signo) {
    pid_t pid;
    int stat;

    pid = wait(&stat); //returns without blocking
    printf("child process exits.");
}
```

3.4 Shared Memory

3.4.1 Overview

Common chunk of read/write memory among processes.



3.4.2 Creating

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024 /* make it a 1K shared memory segment */

int main(void)
{

       key_t key;
       int shmid;
       char *data;
       int mode;

       /* make the key: */
       /* The ftok() function uses the identity of the file named
          by the given pathname (which must refer to an existing,
          accessible file) and the least significant 8 bits of
          proj_id (which must be nonzero) to generate a key_t type
          System V IPC key, suitable for use with msgget(2),
          semget(2), or shmget(2). */
       /* The resulting value is the same for all pathnames that
          name the same file, when the same value of proj_id is
          used. The value returned should be different when the
          (simultaneously existing) files or the project IDs
          differ.*/
       if ((key = ftok("test_shm", 'X')) < 0) {
            perror("ftok");
            exit(1);
       }

       /* create the shared memory segment: */
       /* shmget() returns the identifier of the System V shared
          memory segment associated with the value of the argument
          key. It may be used either to obtain the identifier of a
```

```
        previously created shared memory segment (when shmflg is
        zero and key does not have the value IPC_PRIVATE), or to
        create a new set.*/
    /* A new shared memory segment, with size equal to the value
        of size rounded up to a multiple of PAGE_SIZE, is created
        if key has the value IPC_PRIVATE or key isn't
        IPC_PRIVATE, no shared memory segment corresponding to
        key exists, and IPC_CREAT is specified in shmflg.*/
    /* If shmflg specifies both IPC_CREAT and IPC_EXCL and a
        shared memory segment already exists for key, then
        shmget() fails with errno set to EEXIST. (This is
        analogous to the effect of the combination O_CREAT |
        O_EXCL for open(2).) */
    /* 0644 means permissions of owner, group and user */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT |
        IPC_EXCL )) < 0) {
         perror("shmget");
         exit(1);
    }

    return(0);
}
```

3.4.3 Attach and Detach

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024 /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{

    key_t key;
    int shmid;
```

```
char *data;
int mode;


/* make the key: */
if ((key = ftok("test_shm", 'X')) == -1) {
     perror("ftok");
     exit(1);
}


/* connect to the segment. */
/* There's no IPC_CREATE. Because if there was one this
   function would create a new shared memory.*/
if ((shmid = shmget(key, SHM_SIZE, 0644)) == -1) {
     perror("shmget");
     exit(1);
}


/* attach to the segment to get a pointer to it: */
 /* The shmat() function attaches the shared memory segment
     associated with the shared memory identifier specified
     by shmid to the address space of the calling process.*/
 data = shmat(shmid, (void *)0, 0);
 if (data == (char *)(-1)) {
     perror("shmat");
     exit(1);
 }


 /* read or modify the segment, based on the command line: */
 if (argc == 2) {
     printf("writing to segment: \"%s\"\n", argv[1]);
     strncpy(data, argv[1], SHM_SIZE);
 } else
     printf("segment contains: \"%s\"\n", data);


 /* detach from the segment: */
```

```
        if (shmdt(data) == -1) {
            perror("shmdt");
            exit(1);
        }

       return(0);
}
```

3.4.4 Deleting

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024 /* make it a 1K shared memory segment */

int main(void)
{
       key_t key;
       int shmid;
       char *data;
       int mode;

       /* make the key: */
       if ((key = ftok("test_shm", 'X')) == -1) {
            perror("ftok");
            exit(1);
       }

       /* connect to memory segment: */
       if ((shmid = shmget(key, SHM_SIZE, 0644)) == -1) {
            perror("shmget");
            exit(1);
       }

       /* delete he segment */
```

```
        /* IPC_RMID: Remove the shared memory identifier specified
           by shmid from the system and destroy the shared memory
           segment and shmid_ds data structure associated with it.
           IPC_RMID can only be executed by a process that has an
           effective user ID equal to either that of a process with
           appropriate privileges or to the value of shm_perm.cuid
           or shm_perm.uid in the shmid_ds data structure associated
           with shmid.*/
        if( shmctl(shmid, IPC_RMID, NULL) == -1) {
             perror("shmctl");
             exit(1);
        }

        return(0);
}
```

3.4.5 Command

- ipcs: Lists all IPC objects owned by the user
- ipcrm: Removes specific IPC object

4 Threads

4.1 Problem

Want to do multiple tasks Concurrently

- Start two processes
  - fork() is expensive
  - cold-start penalty

Processes may need to talk to each other

- Two different address spaces, so we need to use IPC
  - kernel transitions are expensive
  - May need to copy data from a user to kernel to another user
  - Inter-process Shared memory is a pain to set up

4.2 Solution

4.2.1 Event-driven programming

- Make one process do all the tasks
- Busy loop polls for events and executes tasks for each event
- No IPC needed
- **Length of the busy loop** determines response latency
- Stateful event responses complicate the code

```
while(1)
{
    Check pending events;
    if (event 1) do task 1;
    if (event 2) do task 2;
    // ...
    if (event N) do task N;
}
```

4.2.2 Threads

Multiple threads of execution per process

4.3 Threads

Shared Resource

- virtual address space(code, heap and static data)
- Open descriptors (files, sockets etc)
- Signals and Signal handlers

Non-Shared resources

- Program counter
- Stack, stack pointer

- Registers
- Thread ID
- Errno
- Priority

4.3.1 Address space layout

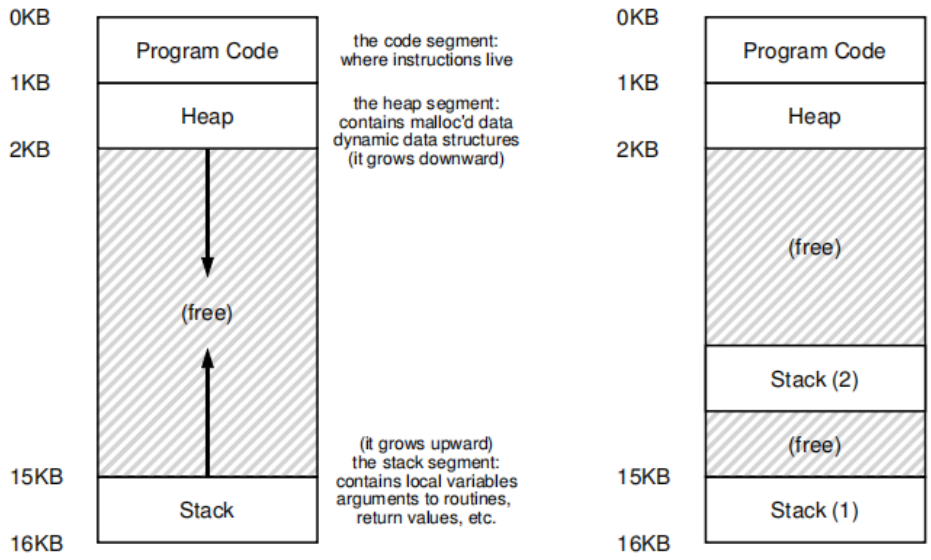


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

4.3.2 Advantages

- Lower inter-thread context switching overhead than processes
- No Inter-process communication
  - Zero data transfer cost between threads
  - Only need inter-thread synchronization
- Threads can be pre-empted at any point
  - Long-running threads are OK

  - As opposed to event-driven tasks that must be short.

- Threads can exploit parallelism, but it depends... more later
- Threads could block without blocking other threads, but it depends ... more later

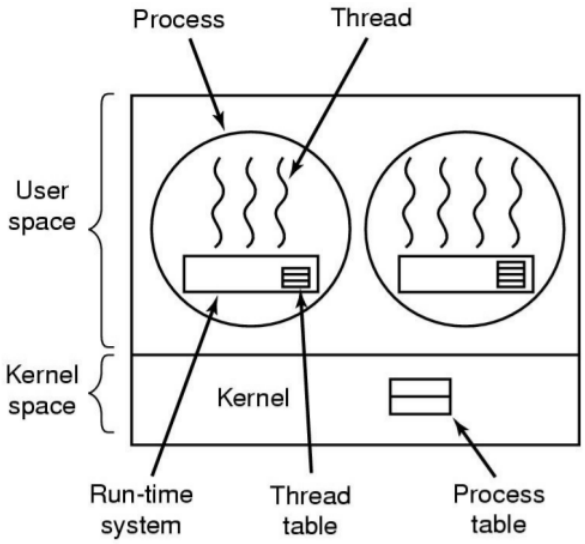
4.3.3 Disadvantages

- Shared State!
  - Global variables are shared between threads.
  - Accidental data changes can cause errors.
- Threads and signals don't mix well
  - Common signal handler for all threads in a process
  - Which thread to signal? Everybody!
  - Royal pain to program correctly.
- Lack of robustness. Crash in one thread will crash the entire process.
- Some library functions may not be thread-safe
  - Library Functions that return pointers to static internal memory. E.g. gethostbyname()
  - Less of a problem these days.

4.3.4 Types of Threads

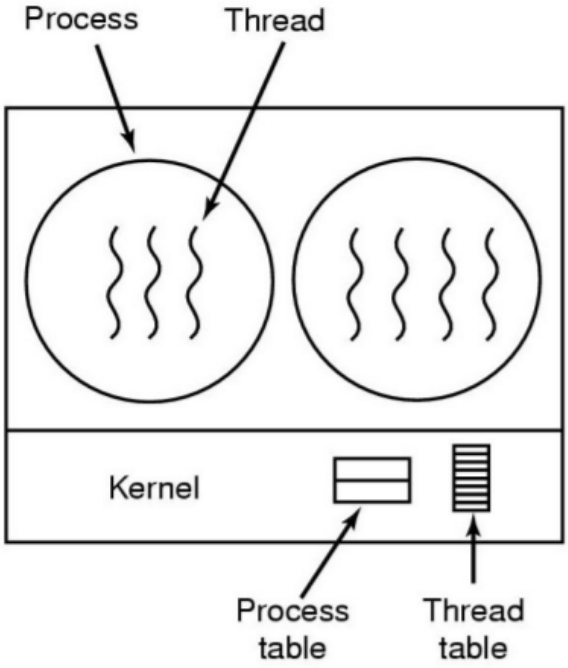
User-level threads

- User-level libraries provide multiple threads,
- OS kernel does not recognize user-level threads
- Threads execute when the process is scheduled

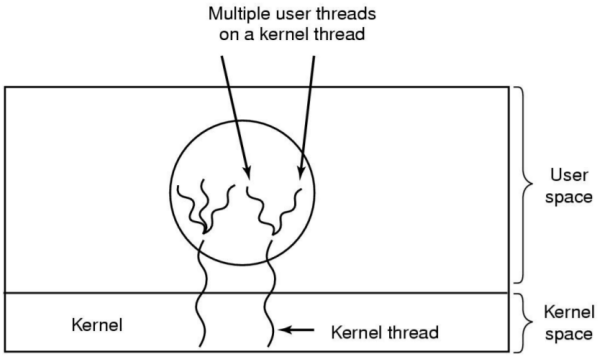


Kernel-level threads

- OS kernel provides multiple threads per process
- Each thread is scheduled independently by the kernel's CPU scheduler



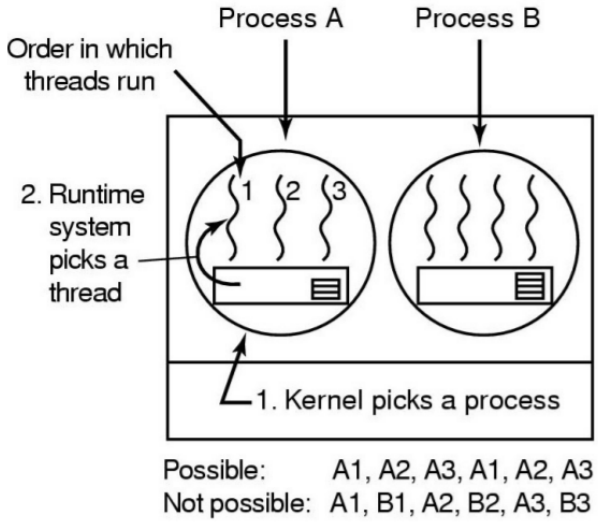
Hybrid Implementations



Multiplexing user-level threads within each kernel- level threads

4.3.5 Scheduling

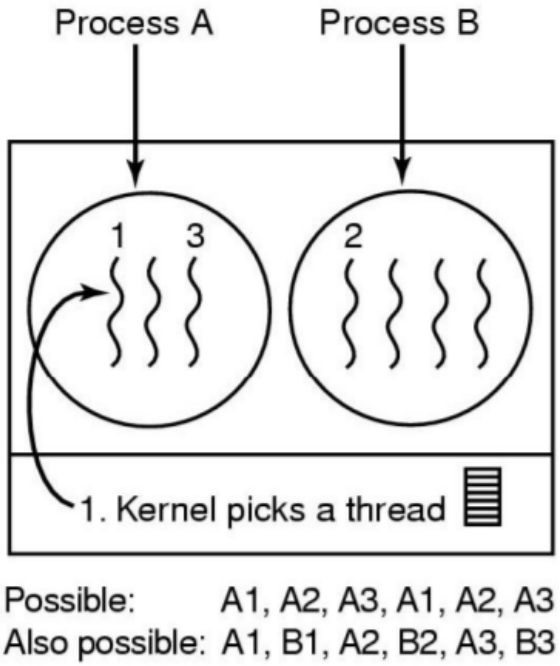
Local Thread Scheduling



- Next thread is picked from among the threads belonging to the *current process*
- Each process gets a timeslice from kernel
- Then the timeslice is *divided up* among the threads within the current process

- Local scheduling can be implemented with either Kernel-level Threads or user-level threads
- Scheduling decision requires only *local knowledge* of threads within the current process.

Global Thread Scheduling



- Next thread to be scheduled is picked up from *ANY* process in the system.
- Timeslice is allocated at the granularity of threads
- Global scheduling can be implemented only with kernel-level threads: for Picking the next thread requires global knowledge of threads in all processes.

4.3.6 Thread Creation and Termination

- Creation

```
int pthread_create( pthread_t * thread, pthread_attr_t
    * attr,
void * (*start_routine)(void *), void * arg);
```

- Two ways to perform thread termination
  - Return from initial function

```
void pthread_exit(void * status)
```

- Waiting for child thread in parent

```
pthread_join()
```

  - equivalent to waitpid

4.3.7 Code

Example

```
// shared counter to be incremented by each thread
int counter = 0;
main()
{
    pthread_t tid[N];
    for (i=0;i<N;i++) {
        /*Create a thread in thread_func routine*/
        Pthread_create(&tid[i], NULL, thread_func, NULL);
    }
    for(i=0;i<N;i++) {
        /* wait for child thread */
        Pthread_join(tid[i], NULL);
    }
    void *thread_func(void *arg)
    {
        /* unprotected code race condition */
        counter = counter + 1;
    }
    return NULL; // thread dies upon return
```

```
}
```

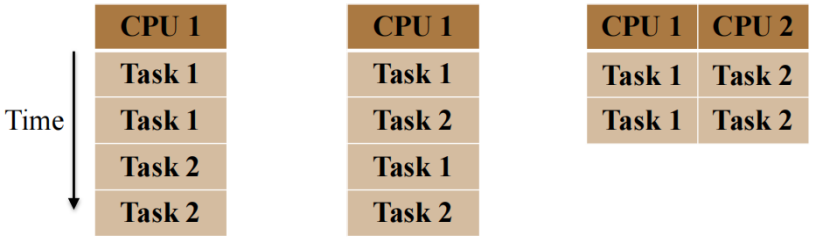
4.3.8 pthread Synchronization Operations

```
// Mutex operation
pthread_mutex_init()
pthread_mutex_lock()
pthread_mutex_unlock ()
pthread_mutex_trylock ()

// Condition variables
pthread_cond_wait ()
pthread_cond_signal ()
pthread_cond_broadcast ()
pthread_cond_timedwait ()
```

5 Concurrency

5.1 Overview

- Sequential: one after another(two tasks executed on one CPU one after another)
- Concurrent: "juggling" many things within a time window(two tasks share a single CPU over time)
- Parallel: do many things simultaneously(two threads executing on two different CPUs simultaneously)



Concurrent tasks must be either

- execute independently
- synchronize the shared resource
  - Shared memory
  - Pipes
  - Signals

5.2 Critical Section

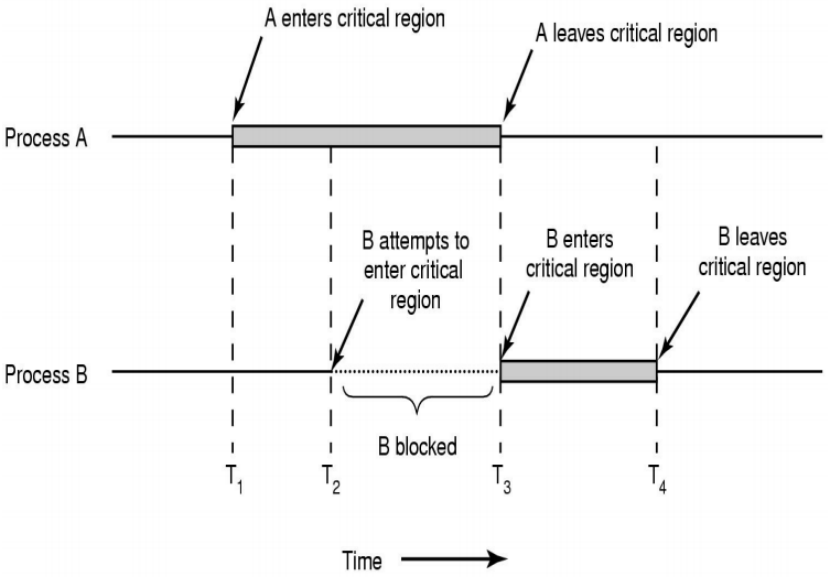
- Definition: A section of code in a concurrent task that **modifies or accesses** a resource shared with another task.
- Example: A piece of code that reads from or writes to a shared memory region

5.3 Race Condition and Deadlocks

- Race Condition: Incorrect behavior of a program due to concurrent execution of critical sections by two or more threads
- Deadlocks: When two or more processes stop making progress **indefinitely** because they are all waiting for each other to do something

5.3.1 Mutual Exclusion

Don't allow two or more processes to execute their critical sections concurrently (on the same resource)



5.3.2 Conditions for correct mutual exclusion

- No two processes are simultaneously in the critical section
- No assumptions are made about speeds or numbers of CPUs
- No process must wait forever to enter its critical section. Waiting forever indicates a *deadlock*
- No process running outside its critical region may block another process running in the critical section

The first two conditions are enforced by the operating system's implementation of locks, but the other two conditions have to be ensured by the programmer using the locks.

5.3.3 Typs of Locks

- Blocking locks
  - Give up CPU till lock becomes available

```
while(lock unavailable)
    yield CPU to others; // or block till lock available
return success;
```

  - Usage:

```
Lock(resource); // Claim a shared resource
Execute Critical Section;//access or modify the shared
    resource
Unlock(resource); // unclaim shared resource
```

  - Advantage: Simple to use. Locking always succeeds. . . ultimately
  - Disadvantage: Blocking duration may be indefinite
    * Process is moved out of "Running" state to "Blocked" state, and return to running will cost much resource
    * Delay in getting back to running state if lock becomes available soon after blocking

- Non-blocking locks
  - Don't block if lock is unavailable

```
if(lock unavailable)
    return failure;
else
    return success
```

  - Usage

```
if(TryLock(resource) == success)
    Execute Critical Section;
    Unlock(resource);
else
    Do something else; // plan B
```

  - Advantage: No unbounded blocking
  - Disadvantage: Need a "plan B" to handle locking failure

- Spin locks

- Don't block. Instead, constantly poll the lock for availability

```
while (lock is unavailable)
continue; // try again
return success;
```

- Usage: Just like blocking locks

```
SpinLock(resource);
Execute Critical Section;
SpinUnlock(resource);
```

- Advantage: Very efficient with short critical sections, if you expect a lock to be released quickly
- Disadvantage:
  * Doesn't yield the CPU and wastes CPU cycles, Bad if critical sections are long: P1 is in the ready queue, but P2 is doing spin with CPU until scheduler interrupts it and give CPU to P1
  * Efficient only if machine has multiple CPUs

5.3.4 Best practices for locking

- **Associate locks with shared resources, NOT code.**
- **Guard each shared resource by a separate lock.**
- **OS cannot enforce these properties**

5.3.5 Deadlock Solution

Deadlock can only be prevented, once it happens, programmers can't solve it by killing the process or enforcing the process to give up the lock.
Right Solution: Lock Ordering

- Sort the locks in a fixed order (say L1 followed by L2)
- Always acquire subset of locks in the sorted order.

5.3.6 Priority Inversion

Conditions

- static priority system
- synchronization between processes

Example

- Definition:
  - Ph – High priority
  - Pm – Medium priority
  - Pl – Low priority
- Procedure
  - Pl acquires a lock L
  - Pl starts executing critical section
  - Ph tries to acquire lock L and blocks
  - Pm becomes "ready" and preempts Pl from the CPU.
  - Pl might never exit critical section if Pm keeps preempting Pl
  - So Ph might never enter critical section
- Problem: A high priority process Ph is blocked waiting for a low priority process Pl, Pl cannot proceed because a medium priority process Pm is executing
- Solution: Priority Inheritance
  - Temporarily increase the priority of Pl to HIGH PRIORITY
  - Pl will be scheduled and will exit critical section quickly
  - Then Ph can execute