Operating System

Yuqiao Meng

2021–9–124

Contents

1 Overview

1.1 What?

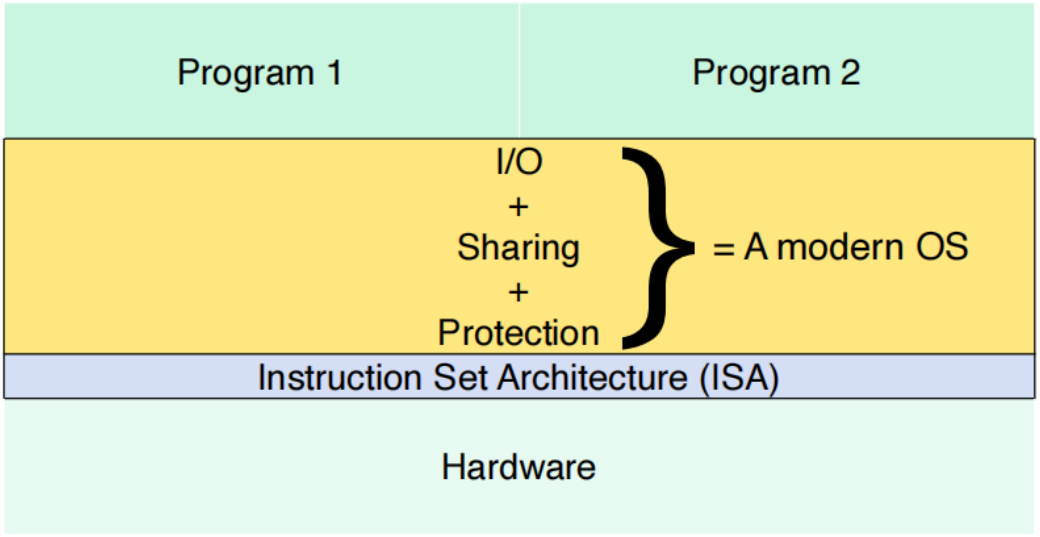
What is an Operating System? What's its reponsibility?

- 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.



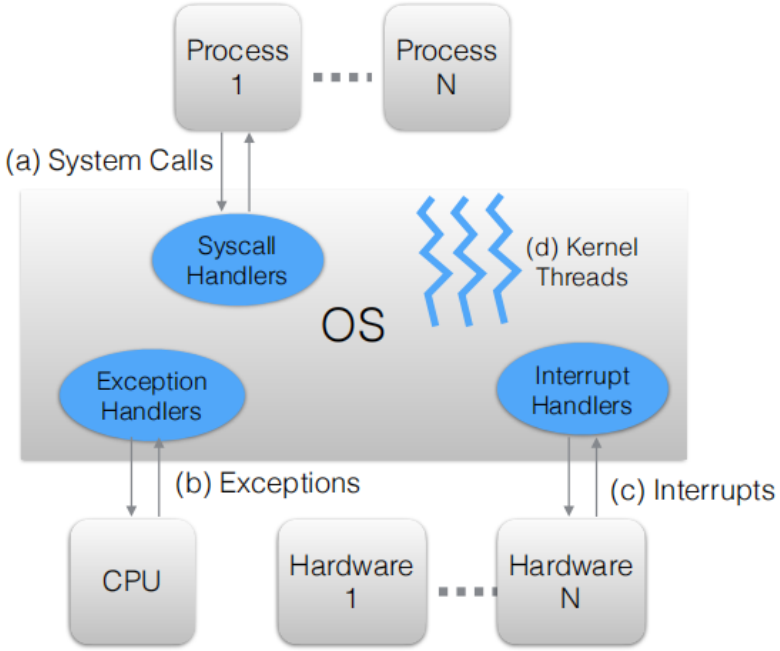
1.3 How

1.3.1 Virtualization

- 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 Virtualization
  - Many(virtual)-to-one(physical): Virtual Machine
  - One-to-many: Memory Virtualization
  - Many-to-many: CPU Virtualization

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.

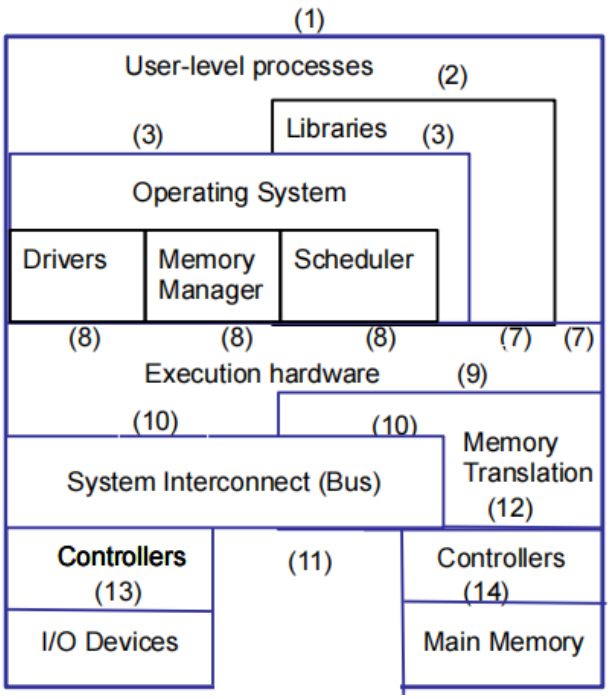


1.4 Interface

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. It's the reason why a program complied on one OS cannot be just moved to another OS.
- 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. ENIAC
- 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. Or FORTRAN

- 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?

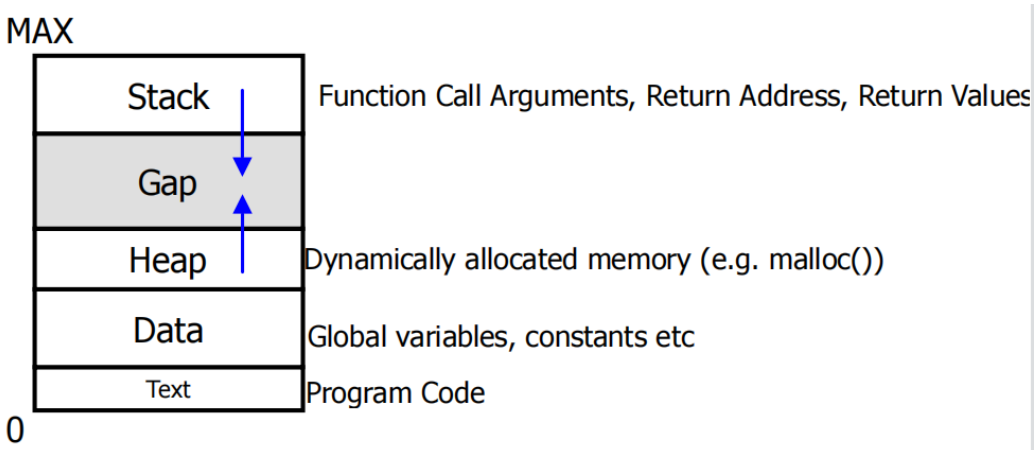
- Program: A passive entity stored in the disk, has static code and static data.
- Process: Actively executing code and the associated static and dynamic data.
- Program is just one component of a process.
- There can be multiple process instances of the same program

2.1.3 Constitution

- Memory space
- Procedure call stack

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

2.1.4 Memory layout



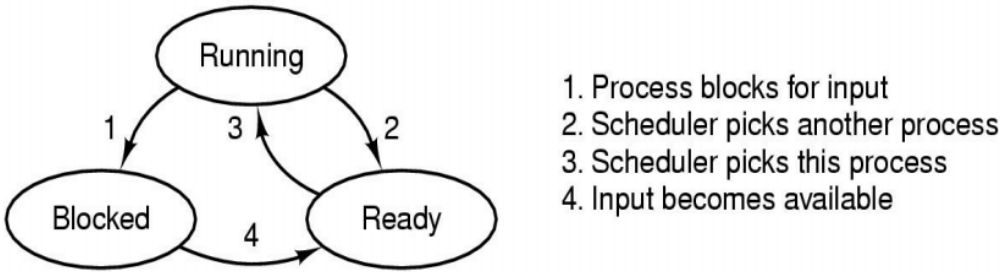
In this picture, Stack and Heap grow toward each other, that's because every process has a limited amount of space, thus let heap and stack grow toward each other from two direction can make the best use of space.

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. **No return if Success. 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: A directional communication mechanism
- Signals: Event notification from one process to another
- Shared momery: Common piece of read/write memory, needs authorization for access

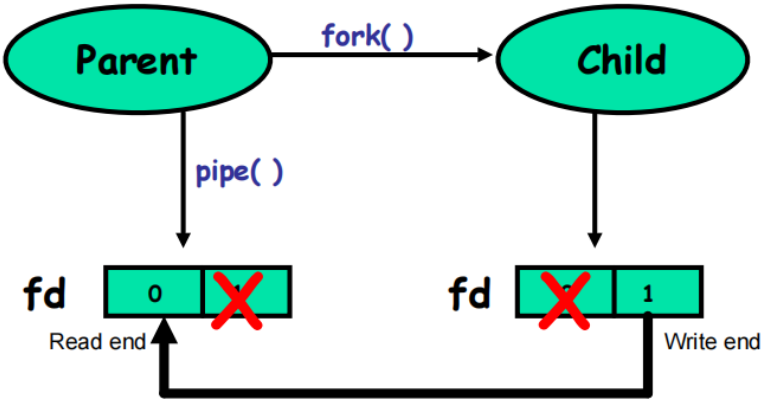
- Parent-child: Command-line arguments, including waitpid(), wait(), exit()
- Reading/modifying common files
- Semaphores: Locking and event signaling mechanism between processes
- Sockets: Not just across the network, but also between processes

3.2 Pipe

3.2.1 Abstraction

Write to one end, read from another.

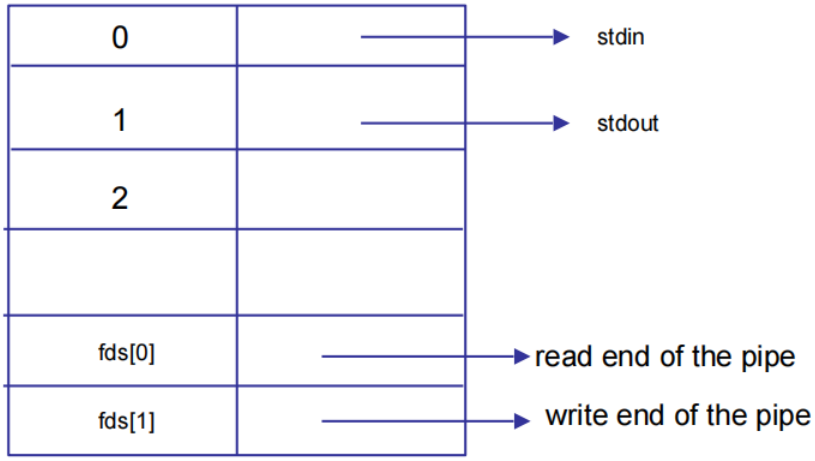
3.2.2 Parent-Child Communication Using Pipe



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)

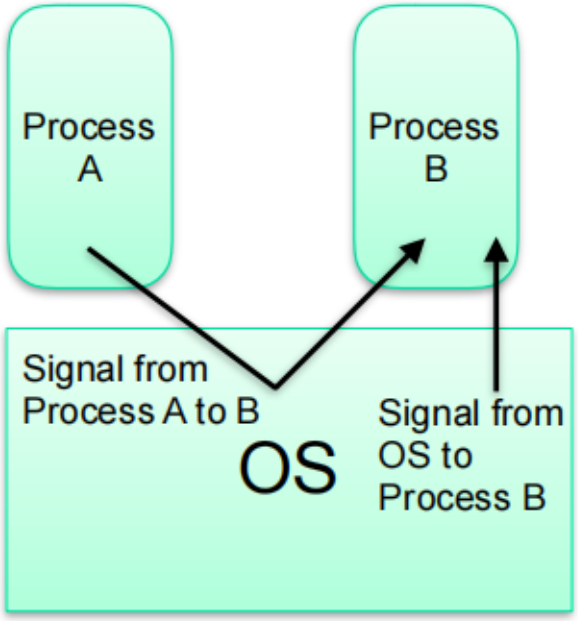
- First check the return value of every read(...)/write(...) system call
- Then
  - Wait to read/write more data
  - Handle any error conditions

```
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 **sigaction()**, 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()

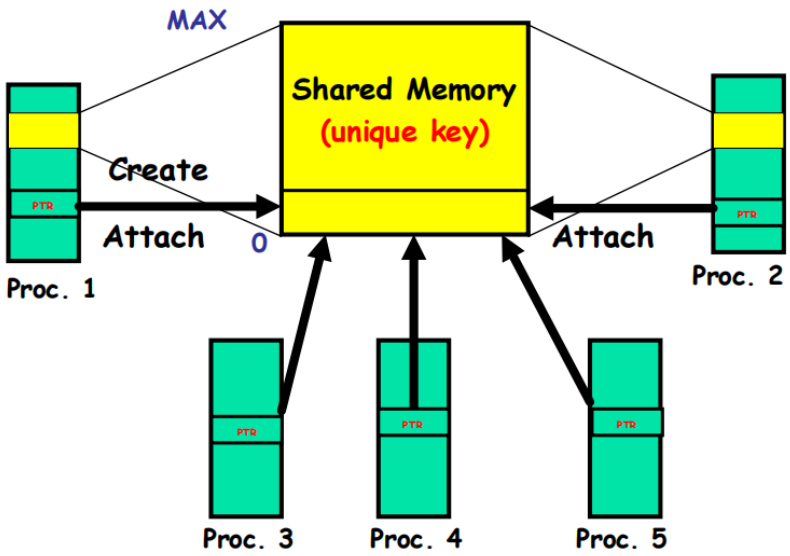
- Parent could install a signal handler for SIGCHLD
- Call wait(...)/waitpid(...)inside the signal handler

```
// 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

```
#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

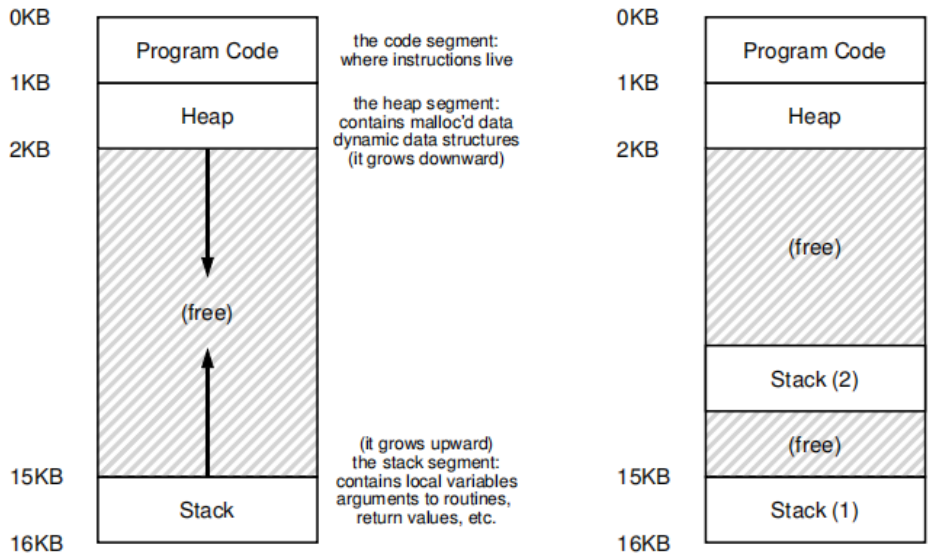


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

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

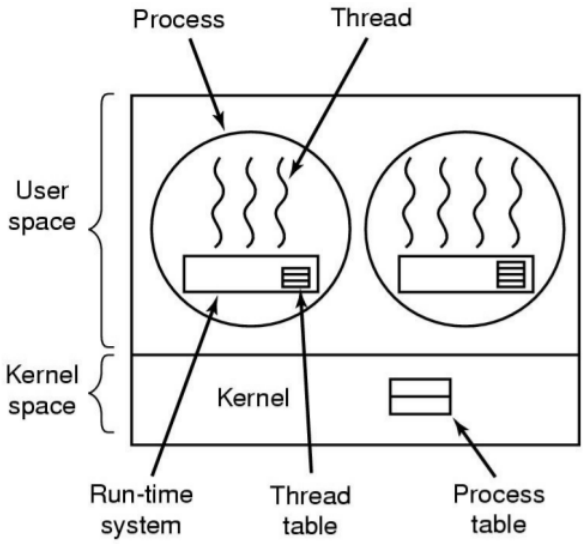
4.3.3 Disadvantages

- 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

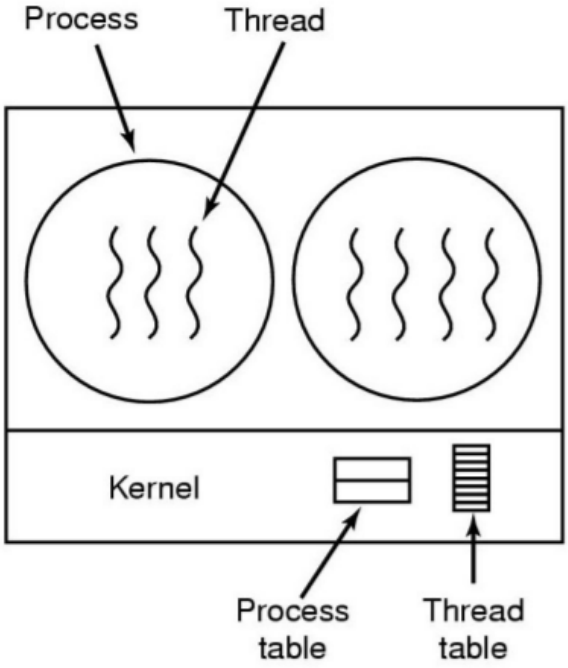
User-level threads

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

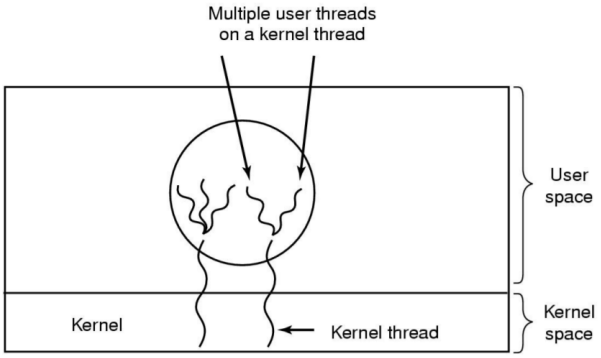


Kernel-level threads

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



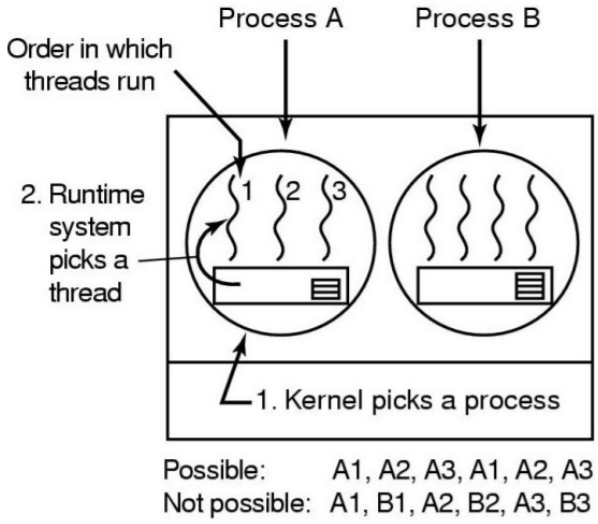
Hybrid Implementations



Multiplexing user-level threads within each kernel- level threads

4.3.5 Scheduling

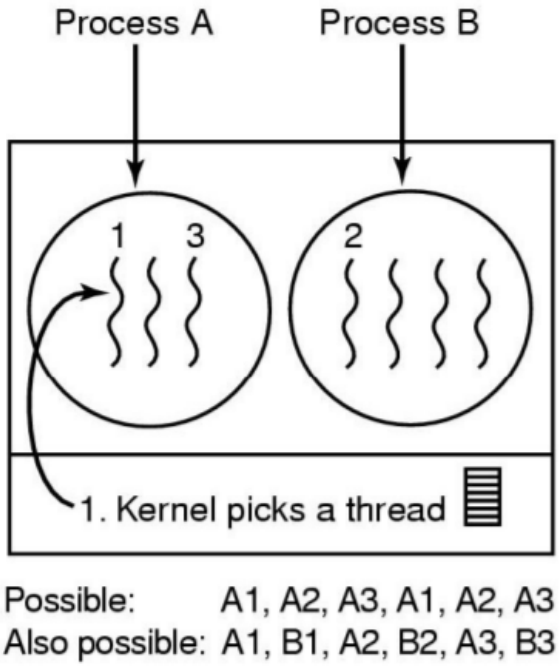
Local Thread Scheduling



- 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
```

```
}
```

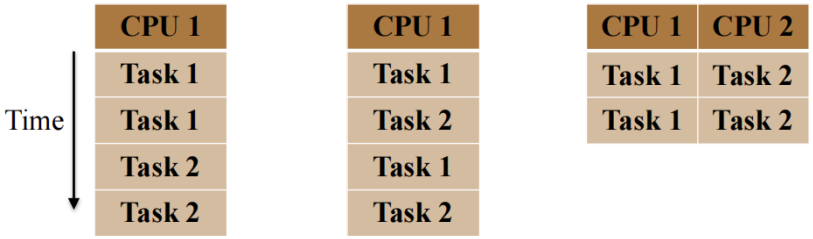
4.3.8 pthread Synchronization Operations

```
// 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

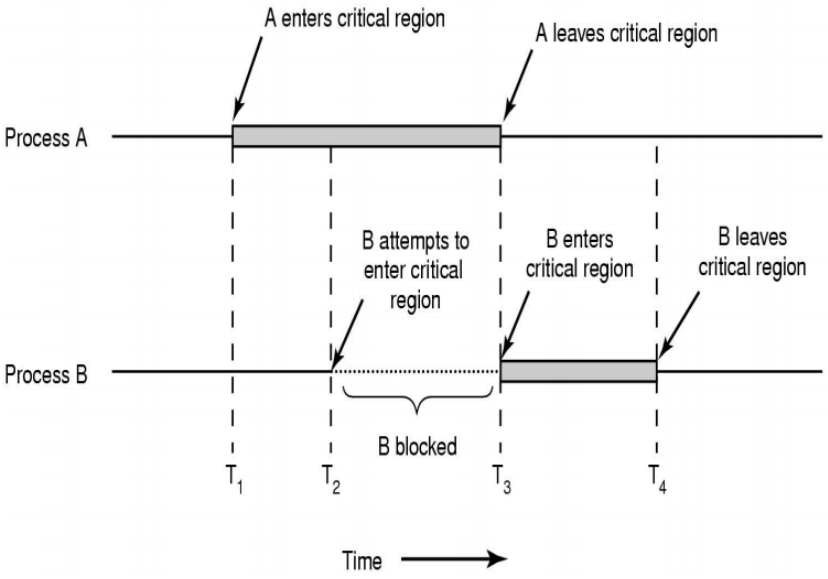
- Definition: A section of code in a concurrent task that **modifies or accesses** a resource shared with another task.
- Example: A piece of code that reads from or writes to a shared memory region

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

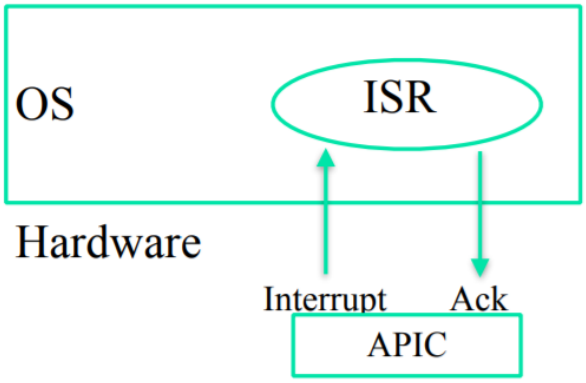
- static priority system
- synchronization between processes

Example

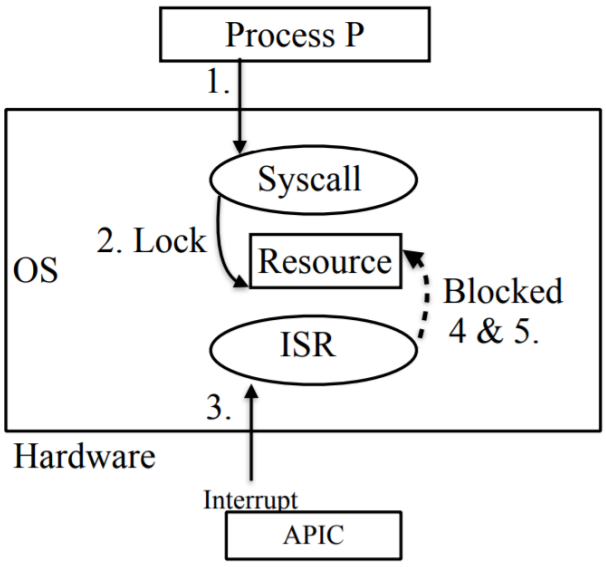
- Definition:
  - Ph – High priority
  - Pm – Medium priority
  - Pl – Low priority
- Procedure
  - Pl acquires a lock L
  - Pl starts executing critical section
  - Ph tries to acquire lock L and blocks
  - Pm becomes "ready" and preempts Pl from the CPU.
  - Pl might never exit critical section if 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

5.3.7 Interrupts and Locks

- Interrupts invoke *interrupt service routines (ISR)* in the kernel
  - ISR must process the interrupt quickly and return: because there may be some other pending interrupts waiting to be delivered.
  - So ISRs must *never block or spin on a lock*.



5.3.8 Interrupts and Deadlocks — Problem



1. P makes a syscall.

2. Syscall acquires lock
3. ISR preempts P* *But P is still in the running state*
4. ISR attempts to lock
5. ISR blocks (since lock is taken)
6. Deadlock!

5.3.9 Interrupts and Deadlocks — Solutions

1. Don't lock in ISR!: Defer any locking work to thread context (softirqs in Linux)
2. If you must lock, use *trylock*() instead of lock() in ISR
   - *trylock*() = if lock is available then get it, else return with error
   - Write code to handle unavailable lock
3. Or disable interrupts in thread T before locking
   - If ISR cannot run when lock is acquired by T, then there's no deadlock.
   - When ISR runs, it assumes that T doesn't have the lock.
   - But, disabling interrupts too long is also not a good idea.

6 Semaphores, Condition Variables, Producer Consumer Problem

6.1 Semaphores

6.1.1 Definition

Can be seen as a non-negative integer

- Semaphore is a fundamental synchronization primitive used for
  - Locking around critical regions
  - Inter-process synchronization

- A semaphore "sem" is a special integer on which only two operations can be performed
  - DOWN(sem)
  - UP(sem)

6.1.2 DOWN(sem) Operation

- If (sem $>$ 0) then
  - Decrements sem by 1
  - The caller continues executing.
  - This is a **successful** down operation.
- If (sem == 0) then
  - Block the caller
  - The caller blocks until another process calls an UP.
  - The blocked process wakes up and tries DOWN again.
  - If it succeeds, then it moves to **ready** state
  - Otherwise it is blocked again till someone calls UP.
  - And so on

6.1.3 UP(sem) Operation

- This operation increments the semaphore sem by 1.
- If the original value of the semaphore was 0, then UP operation wakes up **all processes** that were sleeping on the DOWN(sem) operation.
- All woken up processes compete to perform DOWN(sem) again. Only one of them succeeds and the rest are blocked again.

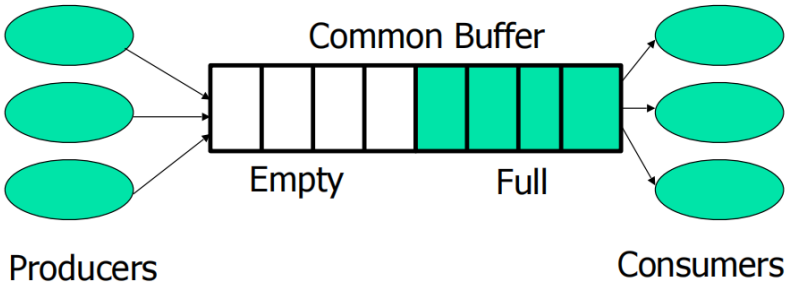
6.1.4 Mutex

A simply binary semaphore

- Used as a LOCK around critical sections
- Locking a mutex means calling Down(mutex)
- Unlocking a semaphore means calling UP(mutex)

6.2 Producer-Consumer Problem

6.2.1 Definition



- Producers and consumers run in concurrent processes.
- Producers produce data and consumers consume data.
- Producer informs consumers when data is available
- Consumer informs producers when a buffer is empty.
- Three types of synchronization needed
  - Locking the buffer to prevent concurrent modification
  - Locking the buffer to prevent concurrent modification and getting
  - Informing the other side that data/buffer is available

6.2.2 Solution

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void){
    int item;
    while(true){
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void){
    int item;
    while(true){
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

6.2.3 Using Semaphore

6.2.4 POSIX interface

- *sem_open*()
- *sem_init*()

- $sem\_wait(), sem_trywait()$
- $sem\_post()$
- $sem\_close()$
- $sem\_destroy()$
- $sem\_getvalue()$
- $sem\_unlink()$ – Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it.

6.3 Monitors and Condition Variables

6.3.1 Definition

```
monitor example
        integer i;
        condition c;

        procedure Function1()

        .
        .           wait(c);
        .
        end;

        procedure Function2()

        .
        .           signal(c);
        .
        end;
end monitor;
```

- Monitor is a collection of critical section procedures (functions): i.e. functions that operate on shared resources
- There's **one global lock** on all procedures in the monitor. Only one procedure can be executed at any time
- **wait(c)** : releases the lock on monitor and puts the calling process to sleep. Automatically re-acquires the lock upon return from wait(c).
- **signal(c)**: wakes up all the processes sleeping on c; the woken processes then compete to obtain lock on the monitor

6.3.2 P-C problem with monitors and condition variables

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

6.4 Atomic Locking – TSL Instruction

- Instruction format: TSL Register, Lock
- Lock
  - Located in memory.
  - Has a value of 0 or 1

- Register: One of CPU registers
- TSL does the following two operations **atomically (as one step)**
  - Register := Lock; // Copy the old value of Lock to Register
  - Lock := 1; // Set the new value of Lock to 1
- TSL is a basic primitive using which other more complex locking mechanisms can be implemented.

Implementation of Mutex Using TSL

```
mutex_lock:
    TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
    CMP REGISTER,#0             | was mutex zero?
    JZE ok                      | if it was zero, mutex was unlocked, so return
    CALL thread_yield           | mutex is busy; schedule another thread
    JMP mutex_lock              | try again later
ok: RET | return to caller; critical region entered


mutex_unlock:
    MOVE MUTEX,#0               | store a 0 in mutex
    RET | return to caller
```

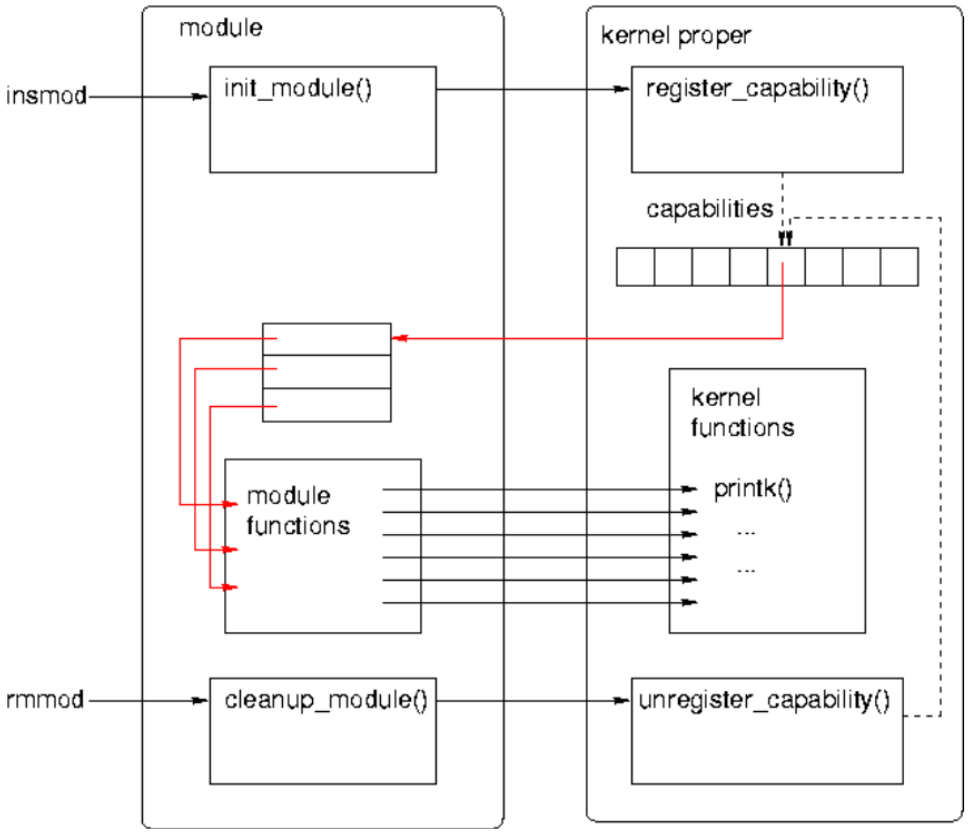
In C-syntax:

```
void Lock(boolean *lock) {
    while (test_and_set(lock) == true);
}
```

7 Kernel Modules

7.1 Definition

- Allow code to be added to the kernel, dynamically
- Only those modules that are needed are loaded. Unload when no longer required - frees up memory and other resources
- Reduces kernel size.
- Enables independent development of drivers for different devices



7.2 Development

7.2.1 Hello World Example

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("DUAL BSD/GPL");
// called when module is installed
int __init hello_init()
{
 printk(KERN_ALERT "mymodule: Hello World!\n");
 return 0;
}
// called when module is removed
void __exit hello_exit()
```

```
{
 printk(KERN_ALERT "mymodule: Goodbye, cruel world!!\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

7.2.2 Compile

- Makefile
  - obj-m := testmod.o
  - For multiple files: module-objs := file1.o file2.o
- Compiling: $ make -C /lib/modules/$(uname -r)/build M=‘pwd‘ modules

7.2.3 Module Utilities

- sudo insmod hello.ko
  - Inserts a module
  - Internally, makes a call to sys_init_module
  - Calls vmalloc() to allocate kernel memory
  - Copies module binary to memory
  - Resolves any kernel references (e.g. printk) via kernel symbol table
  - Calls module's initialization function
- modprobe hello.ko: Same as insmod, except that it also loads any other modules that hello.ko references
- sudo rmmod hello
  - Removes a module
  - Fails if module is still in use
- sudo lsmod
  - Tells what modules are currently loaded
  - Internally reads /proc/modules

7.2.4 Things to Remember

- Modules can call other kernel functions, such as printk, kmalloc, kfree, but only the functions that are EXPORTed by the kernel(using EXPORT(symbol_name))
- Modules (or any kernel code for that matter) cannot call user-space library functions, such as malloc, free, printf etc.
- Modules should not include standard header files, such as stdio.h, stdlib.h, etc.
- Segmentation fault may be harmless in user space, but a kernel fault can crash the entire system
- Version Dependency: Module should be recompiled for each version of kernel that it is linked to

7.2.5 Concurrency Issues

- Many processes could try to access your module concurrently. So different parts of your module may be active at the same time
- Device interrupts can trigger Interrupt Service Routines (ISR), ISRs may access common data that your module uses as well
- Kernel timers can concurrently execute with your module and access common data
- You may have symmetric multi-processor (SMP) system, so multiple processors may be executing your module code simultaneously (not just concurrently).
- Therefore, your module code (and most kernel code, in general) should be re-enterant, Capable of correctly executing correctly in more than one context simultaneously

7.2.6 Error Handling

```
int __init my_init_function(void)
{
    int err;

    /* registration takes a pointer and a name */
    err = register_this(ptr1, "skull");
    if (err) goto fail_this;
    err = register_that(ptr2, "skull");
    if (err) goto fail_that;
    err = register_those(ptr3, "skull");
    if (err) goto fail_those;

    return 0; /* success */

 fail_those: unregister_that(ptr2, "skull");
 fail_that: unregister_this(ptr1, "skull");
 fail_this: return err; /* propagate the error */
}
```

```
void __exit my_cleanup_function(void)
{
    unregister_those(ptr3, "skull");
    unregister_that(ptr2, "skull");
    unregister_this(ptr1, "skull");
    return;
}
```

- In case of failure, undo every registration activity
- But only those that were registered successfully

7.2.7 Module Parameters

- Command line: insmod hellon.ko howmany=10 whom="Class"
- Module code has:

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

7.3 Character devices in Linux

7.3.1 Device Classification

- Character (char) devices
  - byte-stream abstraction(keyboard, mouse)
- Block devices

  - reads/writes in fixed block granularity (hard disks, CD drives)

- Network devices
  - message abstraction, send/receive packets of varying sizes (network interface cards)

- Others
  - USB, SCSI, Firewire, I2O
  - Can (mostly) be used to implement one or more of the above three classes

7.3.2 "Miscellaneous" Devices in Linux

- These are character devices used for simple device drivers.
- All miscellaneous devices share a major number (10).
- But each device gets its own minor number, requested at registration time

7.3.3 Implementing a device driver for a miscellaneous device

1. Declare a device struct

```
static struct miscdevice my_misc_device = {
.minor = MISC_DYNAMIC_MINOR,
.name = "my device",
.fops = &my_fops
};
```

2. Declare the file operations struct

```
static struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .release = my_close,
    .read = my_read,
    ...
    .llseek = noop_llseek
```

```
};
// The function pointers that are not initialized
    above will be assigned some sensible default value
    by the kernel.
```

3. register the device with kernel, usually in the module initialization code

```
static int __init my_module_init()
{
    ...
    misc_register(&my_misc_device);
    ...
}
// And don't forget to unregister the device when
    removing the module
static void __exit my_exit(void)
{
    misc_deregister(&my_misc_device);
    ...
}
```

4. Implement the fops functions

```
static ssize_t my_read(struct file *file, char __user
    * out, size_t size, loff_t * off)
{
    ...
    sprintf(buf, "Hello World\n");
    copy_to_user(out, buf, strlen(buf)+1);
    ...
}
```

5. Warning

   - allocate memory for buf
   - Check if "out" points to a valid user memory location using access_OK()
   - check for errors during copy_to_user()

7.3.4 How do file ops work on character devices

- A file operation on a device file will be handled by the kernel module associated with the device
- Use "open()" system call to open "mydevice" file
  - fd = open("/dev/mydevice", O_RDWR);
  - opens /dev/mydevice device for read and write operation.
  - OS will call my_open() file operation handler in the kernel module which is associated with the device.
  - misc_register(&my_misc_device) in my_module_init() registers the character device. It creates an entry in the "/dev" directory for "mydevice" file and informs the operating system what file-operations handler functions are available for this device.
- Use "read()" system call to read from the "mydevice" file
  - n = read(fd, buffer, size);
  - finally calls the my_read() function passed through the fops structure in your kernel module

7.3.5 Moving data in and out of the Kernel

- copy_to_user()
  - unsigned long copy_to_user (void __user * dst, const void * src, unsigned long n)
  - Copies data from kernel space to user space
  - Returns number of bytes that could not be copied. On success, this will be zero.
  - Checks that dst is writable by calling access_ok on dst with a type of VERIFY_WRITE. If it returns non-zero, copy_to_user proceeds to copy
- copy_from_user()
  - unsigned long copy_from_user (void * dst, const void __user * src, unsigned long n)

- Copies data from user space to kernel
- Returns number of bytes that could not be copied. On success, this will be zero

7.3.6 Memory allocation/deallocation in Kernel

- Memory Allocation:
  - kmalloc(): Allocates physically contiguous memory:

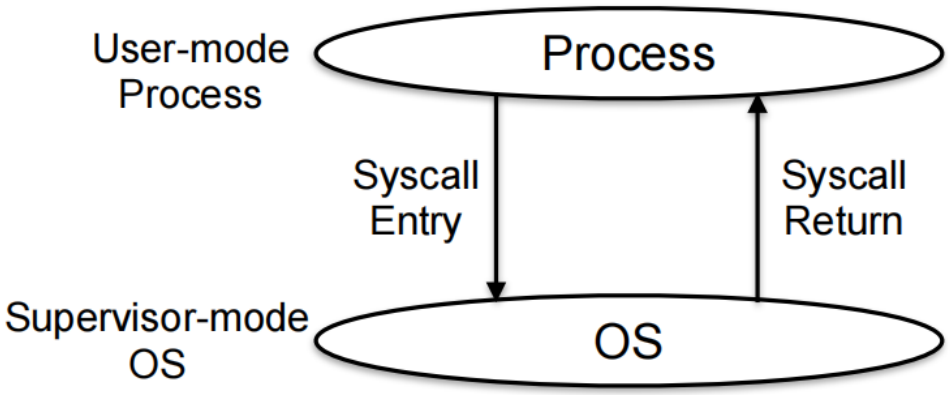
```
void * kmalloc(size_t size, int flags)
```

  - kzalloc(): Allocates memory and sets it to zero
  - vmalloc(): Allocates memory that is virtually contiguous and not necessarily physically contiguous.
- Memory Deallocation: kfree()

8 System Calls

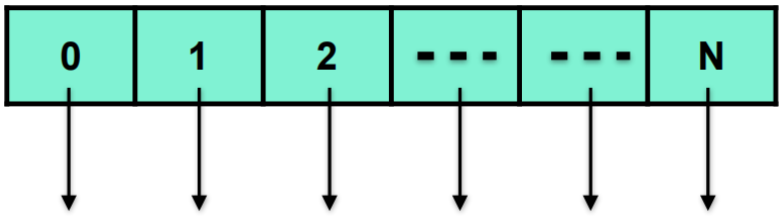
8.1 Definition

- Interface to allow User-level processes to safely invoke OS routines for privileged operations
- Safely transfer control from lower privilege level (user mode) to higher privilege level (supervisor mode), and back



8.1.1 System Call table

- Protected entry points into the kernel for each system call: We don't want application to randomly jump into any part of the OS code
- Syscall table is usually implemented as an array of function pointers, where each function implements one system call
- Syscall table is indexed via system call number



8.1.2 System Call Invocation

1. System calls is invoked via a special CPU instruction: The system call number and arguments passed via CPU registers and optionally stack
2. CPU saves process execution state
3. CPU switches to higher privilege level: jumps to an entry point in OS code
4. OS indexes the system call table using the system call number
5. OS invokes the system call via a function pointer in the system call table
    - For performance reasons, the system call usually executes in the execution context of the calling process, but in privileged mode
    - Some OS may execute the system call in a separate execution context for better security
6. If the syscall involves blokcing I/O, the calling process may block while the I/O completes

7. When syscall completes, the calling process is moved to ready state
8. The saved process state is restored
9. Processor switches back to lower privilege level using SYSEXIT/iret instructions
10. Process returns from the system call and continues.

User process	Invoke syscall using, say, SYSENTER instruction (arguments in registers/stack)
CPU	Switch CPU to supervisor mode. Jump to entry point in kernel.
Kernel	Save process state Lookup Syscall table. Invoke syscall.
Kernel	Optionally Block process if it needs to wait for I/O or other events. Return process to ready state when woken.
Kernel	Restore saved process state SYSEXIT
CPU	Switch CPU to user mode Return to user process
User Process	Return from system call. Continue

8.2 Syscall Usage

- To make it easier to invoke system calls, OS writers normally provide a library that sits between programs and system call interface: Libc, glibc, etc
- This library provides wrapper routines
- Wrappers hide the low-level details of

- Preparing arguments
- Passing arguments to kernel
- Switching to supervisor mode
- Fetching and returning results to application

- Helps to reduce OS dependency and increase portability of programs

8.3 Implementing

8.3.1 Steps in Writing a System Call

1. Create an entry for the system call in the kernel's syscall_table: User processes trapping to the kernel (through SYS_ENTER or int 0x80) find the syscall function by indexing into this table

2. Write the system call code as a kernel function

   - Be careful when reading/writing to user-space
   - Use copy_to_user() or copy_from_user() routines. These perform sanity checks.

3. Implement a user-level wrapper to invoke your system call: Hides the complexity of making a system call from user applications.

8.3.2 Code

1. Create a sys_call_table entry (for 64-bit x86 machines):Syscall table initialized in arch/x86/entry/syscall_64.c

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
...
309 common getcpu sys_getcpu
310 64 process_vm_readv sys_process_vm_readv
311 64 process_vm_writev sys_process_vm_writev
```

```
312 common kcmp sys_kcmp
313 common foo sys_foo
```

2. Write the system call handler

   - System call with no arguments and integer return value

```
SYSCALL_DEFINE0(foo){
    printk (KERN ALERT "sys_foo: pid is %d\n",
        current->pid);
    return current->pid;
}
```

   - Syscall with one primitive argument

```
SYSCALL_DEFINE1(foo, int, arg){
    printk (KERN ALERT "sys_foo: Argument is
        %d\n", arg);
    return arg;
}
```

   - To see system log:
     - dmesg
     - less /var/log/kern.log

Verifying argument passed by user space

```
SYSCALL_DEFINE1(close, unsigned int, fd)
{
    struct file * filp;
    struct files_struct *files = current->files;
    struct fdtable *fdt;
    spin_lock(&files->file_lock);
    fdt = files_fdtable(files);

    if (fd >= fdt->max_fds)
        goto out_unlock;
    filp = fdt->fd[fd];
    if (!filp)
        goto out_unlock;
    …
out_unlock:
    spin_unlock(&files->file_lock);
    return -EBADF;
}
```

- Call-by-reference argument
  - User-space pointer sent as argument.
  - Data to be copied back using the pointer.

```
SYSCALL_DEFINE3( read, unsigned int, fd,
        char __user *, buf,  size_t, count)
{
        …
    if( !access_ok( VERIFY_WRITE, buf, count))
        return –EFAULT;
        …
}
```

3. Invoke syscall handler from user space

   - Use the syscall(...) library function.
   - For instance, for a no-argument system call named foo(), you'll call

```
ret = syscall(__NR_sys_foo);
```

   - For a 1 argument system call named foo(arg), you call

```
ret = syscall(__NR_sys_foo, arg);
```

   - and so on for 2, 3, 4 arguments etc.