CS350 - Operating Systems Part

Topics:

- Brief history of operating systems
- Processes and threads
- Interprocess communication
- Deadlocks
- Files
- Memory

Introduction

What is an Operating System?

Software that acts as an intermediary between the hardware and application programs.

An operating system has two main functions:

1. A manager for access to hardware resources. CPU, memory space, disk space, I/O devices. Protection, fairness, and efficiency are important.
2. To provide abstractions of the hardware at a much higher level than the hardware itself. The file is a much more usable storage abstraction than the disk.

   These abstractions also hide the peculiarities of particular devices (e.g different size disks).

Introduction (contd)

Two general goals of an operating system:

- Convenience for the users
- Efficiency and fairness in resource utilization

An operating system provides a *virtual machine*.

- CS 350 emphasizes the abstractions provided by an operating system - processes, files, special files, communication, etc.
- CS 450, CS 451 emphasize the resource management techniques required in an operating system.

Operating system services

The OS provides services to both the users and the underlying system.

User side:

- program execution
- file manipulation
- (I/O) operations
- communications
- error detection

System side:

- resource allocation
- accounting
- protection

Operating system boundaries

Usually, the OS is taken to comprise all software that runs in supervisor mode.

Other pieces of system software (shells, file manipulation utilities, compilers, etc) are not considered part of the operating system.

Many recent OS have moved many traditional OS components (such as significant parts of the file and process sub-systems) into programs that run in user mode. These components are still regarded as part of the OS, but this trend has blurred the boundary between the OS and other software.

History of operating systems

Because of the close relationship between OS and hardware, OS generations are closely related to hardware generations.

- Babbage's analytical engine (design)—no OS.
- 1st generation (1945–1955). Vacuum tube-based. Plugboards and (later) card input. No OS.
- 2nd generation (1955–1965). Transistor-based. Batch operating systems—tapes, cards and printers. FMS, IBSYS.

The Third Generation (1965–1980)

(Small scale) integrated circuits.

IBM 360 series and OS/360.

(Batch) multiprogramming OS—multiple jobs each in a memory partition. Spooling.

Timesharing OS (terminals). MULTICS—influential.

Mini-computers—PDP range.

Unix.

The 4th Generation (1980–1990)

LSI, VLSI—personal computers, workstations.

MS-DOS and Unix.

Cheap microprocessors—multiprocessor systems (loosely- and tightly-coupled).

Network and distributed OS.

Over the decades, the emphasis has changed from having the OS maximise use of very expensive hardware, to having it provide good abstractions and rapid interactive response.

Unix history

"UNICS" Timesharing OS. Developed 69–73 on a PDP 7 at AT&T Bell Labs. Rewritten in C; ported to PDP 11. Versions 6 (76) and 7 (78) widely distributed. AT&T commercial releases—Systems III (82) and V (83–).

Berkeley (BSD, 78–). Virtual memory, sockets and TCP/IP, vi, csh, improved file system.

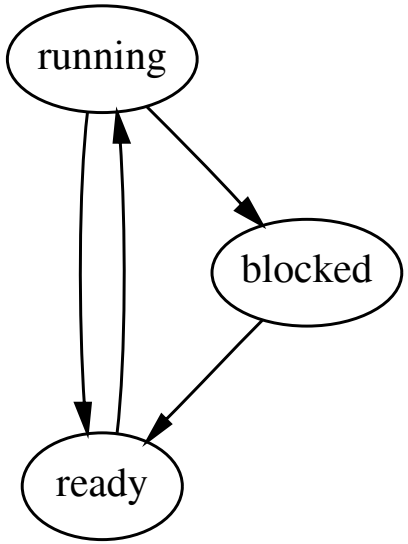
SunOS started as a BSD derivative. Many networking features added (NFS); consider cross fertilisation with other Unix variants.

Linux started as the personal project of Linus Torvalds in 1991. Through co-operative development, it has snowballed into a sophisticated operating system (whose source code is free!).

Highly portable—many derivatives. Standardisation efforts—POSIX, X/Open.

Processes

A running program. Has three main states: *running* - currently using the CPU; *ready* - temporarily suspended by the CPU to let others run; and *blocked* (also called sleeping) - unable to run, waiting for an external event (eg. I/O)



Process Attributes and Process Tables

Processes have:

- a process ID (pid)
- a parent process ID (ppid)
- a user ID (uid).
- a program counter (PC),
- a current size,
- a command name,
- a current working directory (cwd),
- an environment,
- other stuff.

Information about a process in maintained in a data structure called **process table**.

Some of these attributes can be seen by using the `top` and `ps` commands.

Operations on a Process

Operations that can be performed on a process include:

- create a process
- terminate a process
- suspend a process
- resume a process
- modify process's priority
- block a process
- wake-up a process
- dispatch a process

Process Trees

A process can create a new process. The creating process is called a **parent process** while the new process is called a **child process**. The child can in turn create new processes. The result is a hierarchy of process called a **process tree**.

Creating a process: the `fork()` system call.

After `fork`, the new process is a copy of the parent. Both processes continue execution after the `fork`. The only difference is in the return value of the `fork` system call.

```
#include <stdio.h>

main() {
  int pid;

  if ((pid = fork()) == 0) {
    printf("I am the child\n");
  } else {
    printf("I am the parent\n");
  }
}
```


Executing other programs

The members of the `exec` family of system calls overlay the current process with another.

Six versions of `exec`, they differ in whether the arguments are supplied as an array or list, whether the current environment is transferred or not, and whether the path is searched or not.

`execlp` and `execvp` are most often used. Both pass the current environment to the new program, and automatically search the `PATH` environment variable.

The exec calls

`execlp` takes its argument as a list, it is used when the number of arguments is known before hand (at compile time):

```
char *an_arg;
execlp("/bin/cat","cat", an_arg, (char *) NULL);
```

`execvp` takes its argument as an array, it is used when the number of arguments is unkown at compile time:

```
char *the_args[];
execvp(the_args[0], the_args);
```


Waiting and ending

A process can `wait` on its children. Child processes that are finished but not waited for become *zombies*. If a parent terminates without waiting for a child, the child becomes an orphan, and is adopted by the `init` process (pid = 1). `init` waits periodically for children, so oprhan zombies are eventually removed.

Ending a process:

1. `return` from the main function.
2. Finishing the main function. ("Falling off the end")
3. Using the `exit` system call. The integer argument is the return value of the program.

Multitasking

Multitasking is allowing more than one process at a time, so that the system "feels" like there are multiple CPUs.

Co-operative multitasking: A process decides when it needs a break, and lets the operating system give control to another process.

Pre-emptive multi-tasking: Processes are interrupted periodically by the operating system to allow other processes to execute. Typically, a process will be interrupted at least every 100ms.

A process can also be blocked if it is waiting for I/O (e.g. text input) or a resource that is currently busy.

The shell

The operating system is the code that carries out system calls. The OS shell is the command intepreter. It provides an interface between the user and the OS. The shell is started up whenever a user logs-in. The shell is *not* part of the operating system, but makes heavy use of its features.

There are many shells for Unix, some of which are: sh,bash,ash,ksh,csh,tcsh,zsh,tclsh,wish

All of the above shells have the ability to run commands, and write scripts. Most have the ability of file redirection and pipes.

When a simple command is typed in the shell (e.g. `date`), the shell `fork`s, then the child `exec`s the date command, and the parent `wait`s for the child to finish. When the child has finished, the shell prints out the prompt, and tries to read more input.

The shell contd.

Redirection to a file (`date > my.file`) is done by first opening `my.file`, then using the `dup2` system call to copy the file descriptor for `my.file` to the file descriptor for `stdout`. Redirection from a file `sort < my.file` is done simliarly with `stdin`.

Pipes between programs are achieved by connecting the output of one program to the input of the other. Example, the pipe `ls | sort`) is acheived by connecting the standard output of `ls` to the standard input of `sort`. This has the same effect as `ls > temp; sort < temp; rm temp`.

A process can be run in the background using the ampersand (`&`) character (`morph3d &`). In this case, the shell `fork`s and the child `exec`s `morph3d` as usual, but the parent doesn't `wait` for the child to finish. Instead, it prints the prompt immediately, and accepts any input.

Operating System Calls

Operating system calls (usually just called "system calls") are the programmer's interface to the operating system.

UNIX typically has about 200 system calls. These often look similar to library calls, which may or may not make system calls.

Generally, if an error occurs while executing a system call, the system call returns the value `-1`, and sets the global variable `errno` to some appropriate value.

There is a C library function (in `<stdio.h>`) called `perror` which takes a string, and prints out that string along with an appropriate message according to `errno`.

System call errors

There are about 120 possible values for `errno`. These can be seen in the file: `/usr/include/sys/errno.h`.

```
#define EPERM   1  /* Not super-user
#define ENOENT  2  /* No such file or directory
#define ESRCH   3  /* No such process
#define EINTR   4  /* interrupted system call
#define EIO     5  /* I/O error
#define ENXIO   6  /* No such device or address
#define E2BIG   7  /* Arg list too long
#define ENOEXEC 8  /* Exec format error
#define EBADF   9  /* Bad file number
#define ECHILD  10 /* No children
```


Interprocess Communication

Independent process - one that cannot affect or be affected by other executing processes

Cooperating process - one that can affect or be affected by other executing processes

Why process co-operation?

- Need access to a shared resource. (printer)
- Share some workload between processes. (large parallelizable jobs)
- Give instructions to a process. (CTRL-C)
- Pass information to a process. (window size has changed)

Cooperating processes (contd)

Cooperating processes require:

- a mechanism for communication between them
- synchronization between processes - to ensure data consistency

Signals

Signals can be received by a process. They are generated when some (unusual) event requires attention.

They can be generated from various sources:

- **Hardware**, such as divide by zero.
- **Operating System**, such as notifying that file size limit is exceeded.
- **Other Processes**, such as a child process notifying its parent that it has terminated.
- **User**, such as pressing CTRL-Z or CTRL-C, or using the `kill` command.

Actions on Signals

The receiving process can do one of three things upon receiving a signal.

1. **Default**. Do the default action for the signal. For *most* signals, this causes the process to terminate.
2. **Ignore**. Ignore the signal. This can't be done for two of the signals: SIGSTOP, which stops the process from executing, and SIGKILL, which kills the process.
3. **Catch**. Catch the signal. This also can't be done for SIGSTOP and SIGKILL. When a process catches a signal, it executes a special signal handling routine.

Signal numbers

The number of signals is limited. No information can be sent with a signal. The list of signals can be shown by the command `kill -l`.

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGIOT	7) SIGEMT	8) SIGFPE
9) SIGKILL	10) SIGBUS	11) SIGSEGV	12) SIGSYS
13) SIGPIPE	14) SIGALRM	15) SIGTERM	16) SIGUSR1
17) SIGUSR2	18) SIGCHLD	19) SIGPWR	20) SIGWINCH
21) SIGURG	22) SIGIO	23) SIGSTOP	24) SIGTSTP
25) SIGCONT	26) SIGTTIN	27) SIGTTOU	28) SIGVTALRM
29) SIGPROF	30) SIGXCPU	31) SIGXFS	

Sending signals

Within a program, the `raise` system call is used to send signals to yourself.

The `kill` system call is used to send signals to a specified process.

The `alarm` system call sends the `SIGALRM` system call to itself after a specified number of real seconds.


```
/* signal example */
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <stdio.h>

char handmsg[] = "I found ^c\n";
void catch_ctrl_c(int signo)
{
   write(STDERR_FILENO, handmsg, strlen(handmsg));
}

void main(void)
{
   struct sigaction act;

   act.sa_handler = catch_ctrl_c;
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;
   if (sigaction(SIGINT, &act, NULL) < 0)
      perror("Error setting signal handler");
   sleep(1000);
   sleep(1000);
   sleep(1000);
}
```

Signal Summary

- Signals provide a basic communication technique.
- They do not carry any information.
- Participating processes must know each others process IDs.
- The number of signals is limited.
- Cooperating processes must agree on the meaning of each signal.
- No easy way for the sending process to know if its signal was received.
- Signal manipulation can be tricky.

Semaphores and Shared Memory

```
shared int account;

/* deposit with race condition */
void deposit(int money) {

    int balance = account;
    account = balance + money;
}
```

Don't want to be interrupted.

Need mutual exclusion for critical section

More race conditions

```
shared int lock = 0;
shared int account;

/* non-solution for race condition */
void deposit(int money) {

    int balance;

    while (lock == 1) ; /* busy wait */

    lock = 1;

    balance = account;
    account = balance + money;

    lock = 0;
}
```


critical-section problem

Solutions to the *critical-section problem* must satisfy the following:

- **Mutual exclusion**: At most one process in the critical section at any time.
- **Progress**: If no process is executing its critical section, a process that wishes to enter can get in.
- **Bounded waiting**: No process is postponed indefinitely. There must be a limit to the number of times a process may enter a critical section if another process is waiting to get in.

Methods for mutual exclusion (with busy waiting)

- Disabling interupts
- Lock variables
- Strict alternation (round-robin)
- TSL instruction (hardware)

Problems:

- Busy waiting wastes CPU time
- Priority inversion problem

Semaphores

Semaphores can be used to protect critical sections.

Down operation: decrements semaphore, or puts process to sleep if the semaphore is zero.

Up operation: increments semaphore. Wakes up a process if it is sleeping on it.

The semaphore operations are *atomic*, meaning they can't be interrupted.

Semaphore example

```
shared binary semaphore mutex = 1;

/* semaphore solves race condition */
void deposit(int money) {

    int balance;

    down(&mutex);

    int balance = account;
    account = balance + money;

    up(&mutex);
}
```


Semaphores for synchronization

Can also use semaphores for synchronization (e.g. producer–consumer)

```
INITIALIZATION:
        shared binary semaphore mutex = 1;
        shared counting semaphore empty = MAX;
        shared counting semaphore full = 0;
        shared anytype buffer[MAX];
        shared int in = 0, out = 0, count = 0;
PRODUCER :
        anytype item;
        repeat {
                /* produce something */
                item = produce();

                /* wait for an empty space */
                down(empty);

                /* store the item */
                down(mutex);
                buffer[in] = item;
                in = in + 1 mod MAX;
                count = count + 1;
                up(mutex);

                /* report the new full slot */
                up(full);
        } until done;
```

Semaphore example contd.

```
CONSUMER:
        anytype item;
        repeat {
                /* wait for a stored item */
                down(full);

                /* remove the item */
                down(mutex);
                item = buffer[out];
                out = out + 1 mod MAX;
                count = count - 1;
                up(mutex);

                /* report the new empty slot */
                up(empty);

                /* consume it */
                consume(item);
        } until done;
```


Semaphores and UNIX

There are two main implementations of semaphores: POSIX 1003.1b and System V. POSIX implementation of both is simpler, but System V is more widely used. There are also Xenix semphores.

Problems with semaphores: deadlock, all the processes must follow the rules.

Shared memory and UNIX

There are two main implementations of shared memory: POSIX 1003.1b and System V. POSIX implementations are, but System V is more widely used.

The child of a process will inherit all the open files of the parent, (at the time of the `fork`). This can be used to share files between processes. Files can be memory-mapped (using `mmap`), thus creating shared memory between two processes.

The `mmap` system call has the advantage that the contents resides in a file, so it can be used between processes that exist at different times.

Shared memory is very fast. But care must be taken with synchronisation. Using `mmap` is the way to go, as it is less complex and more portable than POSIX or System V shared memory.

Message Passing

Message passing consists of two primitives: `send` (or `write`), and `receive` (or `read`).

There are many variations:

- Buffered and unbuffered.
- Blocking and non-blocking.
- Message boundary preserving and message boundary ignoring.
- half duplex (reading or writing, but not both) and full duplex (reading and writing).
- Reliable and unreliable.

Some issues in Message Passing

Message passing can be used for

- process synchronization and
- interprocess communication.

However, the use of message passing calls for consideration of certain issues, such as:

- Acknowledgements and re-transmissions
- Naming and adressing
- Authetincation
- Performance

Pipes in UNIX

Buffered FIFO. Message boundaries are not preserved. Can be blocking or non-blocking. Usually full-duplex.

Can be named or unnamed.

Unnamed pipes can only be used between two related processes (e.g. child-parent or child-child). They disappear when the processes have finished.

An unnamed pipe is constructed with the `pipe` system call. The syntax is
`int pipe(int fdes[2])` The two file descriptors, `fdes[0]` and `fdes[1]` refer to each end of the pipe. One is used for writing and the other is used for reading.

Named Pipes

Named pipes have an entry in the filesystem. They persist after processes have used them. They can be used between unrelated processes, providing the permissions are set correctly.

A named pipe can be generated from the shell with the `mknod` command. For example:
`mknod aPipe p`
creates a pipe named `aPipe` in the current directory.

This can be written to: `ls > aPipe &`,
... and read from too: `cat < aPipe`.

Named pipes can also be created with the `mknod` system call

popen and pclose

The process of generating a pipe, spawning a child process, organising the file descriptors, and passing command execution information from one process to another via the pipe is quite common.

The standard I/O library provides `popen` and `pclose` to do this.

```
/* uses popen to execute a command
 * and display the output of that
 * command */
#include <stdio.h>
#include <unistd.h>
#include <limits.h>

int main(int argc, char *argv[]) {
  FILE *fin;
  int n;
  char buffer[PIPE_BUF];

  fin = popen(argv[1], "r");

  printf("Output from %s:\n",argv[1]);
  fflush(stdout);

  while ((n = read(fileno(fin),
                   buffer,
                   PIPE_BUF)) > 0) {
    write(1, buffer, n);
  }

  pclose(fin);
  return 0;
}
```


Problems with Pipes

No easy way to determine who the writing process was.

Can't communicate between processes on different machines.

Can get deadlock.

```
/* deadlock example with pipes */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main () {
  int f_des[2];
  static char *s[BUFSIZ];

  if (pipe(f_des) == -1) {
    perror("Pipe");
    exit(2);
  }
  switch (fork()) {
  case -1:
    perror("Fork");
    exit(3);
  case 0:
    write(f_des[1], s, sizeof(s));
    fprintf(stderr,"Child here\n");
    break;
  default:
    write(f_des[1], s, sizeof(s));
    fprintf(stderr,"Parent here\n");
  }
}
```


Sockets in UNIX

Sockets provide an interface to the communication network.

Bufferd FIFO. Can be between unrelated processes. Can be between different machines on a network.

Can have different *domains* of communication:

- UNIX domain: Sockets have actual file names. They can only be used with processes that reside on the same host.
- Internet domain: Allows unrelated processes on different hosts to communicate via the Internet.

Socket types

Two basic types:

STREAM sockets (TCP/IP):

- connection oriented
- reliable.
- communication is with a byte stream
- message boundaries are not preserved.

Socket types (contd)

DATAGRAM sockets (UDP/IP):

- connection-less
- unreliable.
- message boundaries are preserved.
- packets are normally small and fixed in size.

Client-server in UNIX

Socket-based communication is often used for client-server applications. The server provides some service to client programs that connect to it via the socket interface.

Web server and web browser (client).

The client server model is used often. There is a file (`/etc/services`) on most UNIX systems that contains a list of services offered by server programs via the socket interface. The numbers are knwon as *ports*.

/etc/services

```
...
ftp     21/tcp
telnet  23/tcp
smtp    25/tcp  mail
time    37/tcp  timserver
time    37/udp  timserver
name    42/udp  nameserver
whois   43/tcp  nicname     # usually to sri-nic
domain  53/tcp
domain  53/udp
bootps  67/udp              # bootp server
bootpc  68/udp              # bootp client
tftp    69/udp
gopher  70/tcp              # gopher server
finger  79/tcp
http    80/tcp
...
```

Ports under 1024 are reserved. Normal user processes must use ports above 1023.

Sockets from a server's view

The typical sequence for setting up an Internet domain TCP/IP (connection-oriented) socket for a server is:

1. Create the socket with `socket` system call.
2. Assign an address/port pair to the socket with `bind` system call. (In the UNIX domain, a filename is used).
3. Create a queue for incoming connection requests with `listen`.
4. The server is now ready to accept a connection from a client process with the `accept` system call. By default, `accept` will block if there is no request for connection.

5. Once a connection has been made from a client with `accept`, the server can `read` and `write` to the socket. By default, `read` and `write` block if they cannot complete their action.

Often, once a connection has been `accept`ed, the server will `fork`, and the child server will deal exclusively with the client that has connected.

This setup simplifies things somewhat: The parent only needs to accept connections, and fork children. The children only need to respond to requests.

Sockets from a client's view

The client's job is somewhat easier than the server:

1. Create the socket with `socket` system call.
2. Initiate a connection with the server with the `connect` system call.
3. Use `read` and `write` as usual.

Resolving hostnames (i.e. translating mymachine.csee.wvu.edu to 157.182.194.88) is done using the `gethostbyname` system call.

Classical IPC problems

These are basically problems in process coordination and synchronization. There are several variations:

- Producer-consumer problem, (sometimes called the bounded-buffer problem)
- Readers-writers problem for database access
- Dining-philosophers problem for resource contention
- Sleeping-barber problem
- Cigarette-smokers problem
- Observers-reporters problem

Readers-writers problem

A data object is shared among several concurrent processes.

Readers Processes that only want to read the shared object.

Writers Processes that want to update (read and write) the shared object.

Any number of readers may access the data at one time, but writers must have exclusive access.

How do we program the reader and writer?

Approaches to the Readers-Writers problem

A solution to this problem depends on what priorities are required.

1. No reader will be kept waiting unless a writer is already updating the data, (sometimes called *first* reader-writer problem).
2. If a writer is waiting to access the data, no new readers may start reading, (sometimes called *second* reader-writer problem).

Both of these approaches may lead to *starvation*.

Readers-Writers solution for case 1

```
    shared binary semaphore mutex = 1;
    shared binary semaphore db = 1;
    int readcount = 0;
READER:
    down(mutex);
    readcount++;
    if (readcount == 1) then down(db);
    up(mutex);
    CRITICAL REGION where we read
    down(mutex);
    readcount--;
    if (readcount == 0) then up(db);
    up(mutex);
WRITER:
    down(db);
    CRITICAL REGION where we write
    up(db);
```


Dining Philosophers

Five philosophers are seated around a circular table.

There is one fork between each philospher, and a plate of slippery spaghetti in front of each philosopher.

A philosopher needs to grab the two adjacent forks in order to eat, but can only grab one at a time.

Philosophers alternate between eating and thinking.

They only eat for finite periods of time.

Dining Philosophers – poor solution

```
shared binary semaphore fork[5] = 1;

Philosopher(i) {

     while(TRUE) {
          think();
          down(fork[i]);
          down(fork[(i+1)%5]);
          eat();
          up(fork[i]);
          up(fork[(i+1)%5]);
     }
}
```


Dining Philosophers – better solution

Proposed solution: We could modify the algorithm so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left fork, waits for a while, then tries the whole process again.

What if all philosophers start at the same time? The programs continue to run but never make progress: *starvation*.

Why not have each process wait a random amount of time?

Want a solution that always works and can't fail due to an unlikely series of random numbers.

Dining Philosophers solution 1

```
shared binary semaphore fork[5] = 1;
shared binary semaphore mutex = 1;
Philosopher(i) {

     while(TRUE) {
        think();
        down(&mutex);
        down(fork[i]);
        down(fork[(i+1)%5]);
        eat();
        up(fork[i]);
        up(fork[(i+1)%5]);
        up(&mutex);
     }
}
```

Only one philosopher can eat at a time.

What if we had 100 philosophers? Up to 50 should be able to eat at once.


```
Dining philosophers solution 2:
        int state[n];
        shared binary semaphore mutex = 1;
        shared binary semaphore s[N];
            void philosopher (int i) {
                  while ( TRUE ) {
                        think();
                        take_forks(i);
                        eat();
                        put_forks(i);
                  }
            }
             void take_forks(int i) {
                  down(&mutex);
                  state[i] = Hungry;
                  test(i);
                  up(&mutex);
                  down( &s[i]);
            }
            void put_forks(int i) {
                  down(&mutex);
                  state[i] = Thinking;
                  test(LEFT);
                  test(RIGHT);
                  up(&mutex);
            }
            void test (int i) {
                  if ( state[i] == Hungry &&
                        state[LEFT] != Eating &&
                        state[RIGHT] != Eating )    {
                      state[i] = Eating;
                      up (&s[i] );
            }
```

Equivalence of IPC Primitives

Under certain conditions (for instance, on a machine with a single CPU), the different interprocess communication primitives are semantically equivalent.

That is, for each or the three basic primitives - semaphores, message passing or monitors, we can use one to implement the other. This further implies that, the synchronization problems solved using one primitive can also be solved using any of the others.

Deadlocks

Resources These are generally objects, (could be hardware, e.g. printer, or software, e.g. data records), to which processes can be given exclusive rights.

Preemptable resource - a resource which can be taken away from the current process holding it, without causing the computation to fail, e.g. CPU and memory

Non-preemtable resource - a resource which cannot be taken away from its current owner without adverse effects on the results, example, printer. Deadlocks usually involve non-preemtable resources.

Event Sequence for resources: 1. Request resource; 2. Use resource; 3. Release resource

Deadlock. A system is in a deadlock if for some set of processes in the system, each process in the set is waiting for an event that can only be caused to occur by another process in the set.

Conditions for deadlock

A deadlock condition can arise if the following four conditions hold simultaneously:

1. **Mutual Exclusion:** At least one resource must be non-sharable (Only one process at a time can use it).
2. **Hold and wait:** Processes currently holding resources granted earlier can request new resources.
3. **No preemption:** Resources given to a process can not be forcibly taken away from the process. A process must give them up voluntarily.
4. **Circular wait:** There must be a circular chain of processes, each of which is waiting for a resource held by the next member of the chain.

Handling deadlocks - The ostrich algorithm

All four conditions must be present for a deadlock to occur. If one is absent, no deadlock is possible.

What should be done if deadlock occurs?

Can use the ostrich algorithm.

This is the most common method of dealing with deadlocks: stick your head in the sand and pretend there is no problem at all.

Works quite well if deadlocks occur infrequently.

Preventing deadlocks can be quite complicated to do well (The dining philosophers problem.)

Allowing deadlocks to occur and then recovering from them is also an option.

Resource Allocation Graph

The resource allocation graph (RAG)is a directed graph that shows the relationship between the processes and resources in the system. An edge from a process to a resource indicates that the process is making a request for the resource, and that the resource has not yet been allocated to the process. For a resource that has successfully been claimed by a process, the edge is from the resource to the process.

Process P requests for resource R: $P \rightarrow R$

Resource R is allocated to P: $R \rightarrow P$

A typical RAG can thus be represented by a set of edges:

E ={ $P1 \rightarrow R1$, $P2 \rightarrow R3$, $R1 \rightarrow P2$, $R2 \rightarrow P2$, $R2 \rightarrow P1$, $R3 \rightarrow P3$ }

Deadlock Detection

Single Instance of the Resources
When there are single instances of each resource type, it is easy to detect deadlocks using the RAG. This is done by detecting cycles in the graph. If a cycle exists, there is a deadlock in the system, otherwise there is no deadlock.

Multiple Instances of Each Resource
When, there are possible multiple instances of each resources, the existence of cycles in the resource allocation graph may not necessarily mean that deadlock exists. A cycle is a necessary but not sufficient condition for deadlocks in such cases. Thus, new conditions may need to be met to detect deadlocks.

Typically, a matrix based method is used, which makes use of different information, such as the existing resource and available resource vectors, and the current allocation and request matrices

Deadlock Detection with Multiple Resources using RAGs

In the case of multiple instances of a resource the following strategy can be used to detect deadlock using the resource allocation graph:

There must exist some cycle in the system.

Let R_{kn} = number of resource R_k available. Then, for there to be a deadlock involving resource R_k, R_k must be involved in at least R_{kn} cycles.

Thus, in the case of multiple instances, the following should be done:

1. Detect ALL the cycles in the graph
2. For each resource in the system (or a cycle), check how many cycles that involve that resource.

3. If a resource with say R_{kn} instances is involved in $< R_{kn}$ number of cycles, then that resource is not involved in a deadlock.
4. If (3) above holds for all the resources in the system (or the cycle), then there is no deadlock.

Deadlock Aviodance

With some advance information about processes' resource need, deadlock can be avoided.

Avioding deadlock requires the system to know if allocating a requested resource will result into a safe or an unsafe state.

Safe State - There exists at least one way in which the system can satisfy all resouce requests. That is, the system can guarantee that all processes will finish.

Unsafe State - There is no guarantee that all request will be satisifed. The system is not yet in a deadlock, but could get into one.

The general method used to determine if a system will be in a safe or unsafe state is the Banker's algorithm. This is however not always applicable because it requires:

- prior knowledge of maximum resource needs,
- fixed number of processes,
- fixed number of resources.

Deadlock Prevention

Generally, deadlocks can be prevented by denying at least one of the conditions for deadlock.

Denying Mutual Exclusion Condition. Depends on the particular resource involved. Could be quite difficult, and expensive. But, mutual exclusion may be necessary for certain operations.

Denying Hold and Wait Condition. Processes reqest all the required resources at once. Resource allocation is done on an all-or-none basis. Problems here include need for advance information on resource needs, and resource under-utilization.

Denying the No Preemtion Condition. If a process holding a certain resourse is denied further resources, the process must relinquish the already held resources. Applicability depends on the type of resource. Results in waste of resources. Could also lead to starvation.

Denying Circular Wait Condition. Order the resources numerically. Request for resources are made in numerical order. E.g. If process A holds resurce R_i, and process B holds resource R_j, and $f(R_j) > f(R_i)$, where f(.) is an ordering function, then process B cannot request for resource R_i. To obtain R_i, B must first relinquish R_j.

As the number of resources increase, finding the appropriate ordering function could become a problem.

We can see that each method for deadlock prevention has its own problems.

Deadlock Recovery

From the above problems, it may be good to still prepare for deadlock. There are some methods for deadlock recovery. Yet, non is particularly attractive.

Preemption - take away some resources from some processes, and allocate them to some other processes, until the deadlock is broken.

There are some issues to consider here:

- Which Victim - how do we choose the process and/or resources to preempt ?

- Rollback - using process checkpoints. For the process to be preempted, roll back to the checkpoint just before the process acquired a resource in a deadlock. If the process tries to aquire the same resoure again, it must wait until it becomes available.
- Starvation - how can we ensure that the resources are not always taken away from the same process?

Termination - kill one or more of the processes. The process(es) chosen may or may not be involved in the deadlock.

Threads

A process has an address space. In its address space a process has the code it is executing (the text), and various pieces of data.

When a program executes, the CPU uses the process program counter value to determine which instruction from the address space to execute next.

The resulting stream of instructions is called the program's *thread of execution*.

Single-threaded process

Process one executes the statements 245, 246, and 247 in a loop.

Its thread of execution can be represented by the sequence:
$245_1, 246_1, 247_1, 245_1, 246_1, 247_1, 245_1, 246_1, 247_1, \ldots$
The subscripts identify the thread of execution.

Process two executes the statements 101, 102, 103, 104, ...

The CPU might execute instructions in the order:
$245_1, 246_1, 247_1, 245_1, 101_2, 102_2, 103_2, 104_2, 105_2, 246_1, 247_1 \ldots$

Context switches occur between 245_1 and 101_2, and between 105_2 and 246_1.

The processor sees the threads of execution interleaved, but the processes see continuous sequences.

Multi-threaded process

We can extend the process model to allow multiple threads of execution within a process.

The sequence of instrucions executed in a process with two threads might be:
$572_{1a}, 573_{1a}, 574_{1a}, 575_{1a}, 412_{1b}, 413_{1b}, 414_{1b}, \ldots$

Threads within a process share everything (open files, variables, memory etc.), except the current point of execution, and the contents of the registers.

Threads are sometimes called *light-weight processes*.

A thread can belong to only one given process. Thus, while processes can be created by different users, threads belong to only one user.

Why threads?

Why use threads if we can use `fork` to create a new process, and use shared memory or some other form of IPC?

Speed.

The context switch between two processes is expensive. It has to replace the context of the current process with another.

This includes:

- The stack.
- Registers.
- Program counter.
- Executable code and memory used for variables.

- Process state.
- I/O status.
- User ID.
- Process ID.
- Any special priveledges.
- Scheduling information.
- Accounting information.
- Memory management information.
- Heaps of other stuff.

Switching between threads of a process only requires a change of the program counter, register set, execution stack, and state.

More about threads

On a multiprocessor machine, threads can run simultaneously. (usually)

Parallelism can be acheived with low overhead.

But, they complicate programming.

All global variables are shared, so synchronisation can be a problem.

System calls may or may not be thread-safe if the kernel is not multi-threaded.

The two basic types of threads are user level threads and kernel-level threads.

User level threads

User-level threads are implemented in a library.

The kernel doesn't know about them (they are independent of the kernel).

The thread library adds some extra code to each system call. (A *wrapper* or *jacket*).

The extra code does thread management, and takes care of system calls that might block (such as `read` or `sleep`), otherwise the calls might block the whole process rather than just the thread that called them.

More on user level threads

Advantage Low overhead: switching between threads is quick.

Problems A greedy thread that does no system calls or library calls won't let the thread management code schedule another thread. The programmer may have to explicitly yeild control at appropriate points.

Can only share resources allocated to their associated process. This effectively limits threads to a single processor, negating one of the prime motivations to use threads.

Kernel level threads

Kernel is aware of all threads.

Each thread is separately schedulable.

Each thread competes for resources on a system-wide basis.

Scheduling kernel-level threads is almost as expensive as processes, but kernel-level threads can use multiple processors.

A kernel-level thread takes about seven times longer to create than a user-level thread. (A `fork` takes about 30 times longer to create than a user-level thread).

User/kernel hybrid threads

Solaris uses a hybrid aproach to threads.

The user writes the program in terms of user-level threads but specifies how many kernel-schedulable entities are associated with the process.

The user-level threads are mapped to the kernel-schedulable entities as the process is running.

In Solaris, a user-level thread is called a thread. Each is supported by a kernel-schedulable entity called a *lightweight process* (LWP). Each LWP is mapped to only one kernel-level thread.

Using POSIX threads

`pthread_create` creates a thread to execute a specified function.

`pthread_exit` causes the calling thread to terminate without the whole process terminating.

`pthread_kill` sends a signal to a specified thread.

`pthread_join` causes the calling thread to wait for the specified thread to exit. This is similar to `waitpid` for processes.

`pthread_self` returns the callers identity (The thread ID).