Concurrency Management

- Several design and management issues raised by the existence of concurrency
- The OS must:
  - Be able to keep track of various processes
  - Allocate and de-allocate resources for each active process
  - Protect the data and physical resources of each process against interference by other processes
  - Ensure that the processes and outputs are independent of the processing speed

Principles of Concurrency and Synchronization

- Concurrency context: multiple applications, structured applications, OS structure
- Principles:
  - Interleaving and overlapping
  - The relative speed of execution of processes cannot be predicted
- Difficulties:
  - Sharing of global resources
  - Difficult for the OS to manage the allocation of resources optimally
  - Difficult to locate programming errors as results are no longer deterministic and reproducible
- Race condition:
  - Occurs when multiple processes or threads read and write data items
  - The final result depends on the order of execution
  - The "loser" of the race is the process that updates last and will determine the final value of the variable

Race Condition Example

- count++ could be implemented as

```
register1 = count    register1 = register1 + 1    count = register1
```

- count– could be implemented as

```
register2 = count    register2 = register2 - 1    count = register2
```

- Interleaving execution with count = 5 initially:
  1. A execute `register1 = count` (so register1 = 5)
  2. A execute `register1 = register1 + 1` (so register1 = 6)
  3. B execute `register2 = count` (so register2 = 5)
  4. B execute `register2 = register2 - 1` (so register2 = 4)
  5. A execute `count = register1` (so count = 6)
  6. B execute `count = register2` (so count = 4)
- Problem: Two processes are at the same time in the same critical region (count)

Key Terms

- Atomic operation: An operation (sequence of instructions) that appears to be indivisible: no other process can see an intermediate state or interrupt the operation; the sequence of instruction is guaranteed to execute as a group or not execute at all; atomicity guarantees isolation from concurrent processes
- Critical section: A section of code within a process that must not be executed while another process is in a corresponding section of code
- Deadlock: Processes that are unable to proceed because each is waiting for one of the others to do something
- Livelock: Two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work
- Mutual exclusion: When one process is in a critical section that accesses some resources no other process may be in a critical section that accesses any of those shared resources
- Race condition: Multiple processes read and write a shared data item and the final result depends on the relative timing of their execution
- Starvation: A runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen

Process Interaction

- Results of one process independent of the action of others
- Timing of process may be affected
- Processes unaware of each other = competition
  - Potential control problems: mutual exclusion, deadlock (renewable resource), starvation
- Processes indirectly aware of each other (e.g., shared object) = cooperation by sharing
  - Potential control problems: mutual exclusion, deadlock (renewable resource), starvation, data coherence
- Processes directly aware of each other (communication primitives available) = cooperation by communication
  - Potential control problems: deadlock (consumable resource), starvation

Mutual Exclusion

```
/* Process P1 */          /* Process P2 */              /* Process Pn */

void P1 () {              void P2 () {                  void Pn () {
  while(true) {             while(true) {                 while(true) {
    // preceding code         // preceding code             // preceding code
    enterCritical(Ra);        enterCritical(Ra);    ...     enterCritical(Ra);
    // critical section       // critical section           // critical section
    exitCritical (Ra);        exitCritical (Ra);           exitCritical (Ra);
    // following code         // following code             // following code
  }                         }                             }
}                         }                             }
```

- Must be enforced
- A process that halts must do so without interfering with other processes
- No deadlock or starvation
- A process must not be denied a critical section when there is no one using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section only for a finite time

Hardware Support for Mutual Exclusion

- Interrupt disabling
  - Guarantees mutual exclusion, but only on uniprocessor systems
  - Degradation in the efficiency of execution
  - Will not work in a multiprocess architecture
- Special machine instruction
  - Atomic compare and swap (or compare and exchange)
    - Comparison between a memory value and a test value
    - If the values are the same then a swap occurs
  - Atomic exchange
    - Exchanges the values of two variables
  - Advantages: applicable generally (any number of processors), simple and easy to verify, supports multiple critical sections (each with its own variable)
  - Disadvantages: busy waiting, possibility of starvation and deadlock

Compare and Swap Mutual Exclusion

```
int bolt;

void P (int i) {
    while (true) {
        // preceding code
        while (compare_and_swap(bolt, 0, 1) == 1)
            /* NOP */;
        // critical section
        bolt = 0;
        // following code
    }
}

void main () {
    bolt = 0;
    // run in parallel P(0), P(1), P(2), ..., P(n)
}
```

EXCHANGE MUTUAL EXCLUSION

```
int bolt;

void P (int i) {
    int keyi = 1;
    while (true) {
        // preceding code
        do exchange(keyi, bolt);
        while (keyi != 0);
        // critical section
        bolt = 0;
        // following code
    }
}

void main () {
    bolt = 0;
    // run in parallel P(0), P(1), P(2), ..., P(n)
}
```


COMMON CONCURRENCY MECHANISMS

Semaphore	An integer variable with only three operations (all atommic) may be performed on a semaphore: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a counting semaphore or a general semaphore.
Binary Semaphore	A semaphore that can only be initialized to 1.
Mutex	Similar to a binary semaphore, but the process that locks the mutex (sets the value to zero) must be the one to unlock it (set the value to 1).
Condition Variable	A data type that is used to block a process or thread until a particular condition becomes true.
Monitor	A programming language construct that encapsulates variables, access procedures and initialization code in an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are critical sections. A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/Messages	A mean for two processes to exchange information. May also be used for synchronization.
Spinlocks	Busy waiting mutual exclusion.

SEMAPHORES

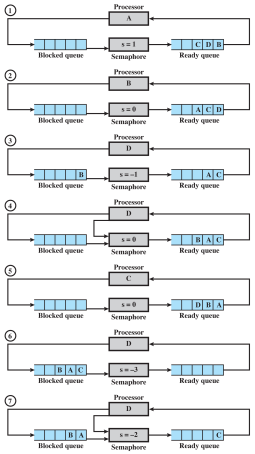
- There is no way to inspect or manipulate semaphores other than via three operations: initialize (with a non-negative integer), wait, and post
- Consequences:
  - No way to know before a process decrements a semaphore whether it will block
  - No way to know which process will proceed when two processes are running concurrently
  - Don't know whether another process is waiting (the number of unblocked processes may be zero or one)

```
struct semaphore { int count; queueType queue; };

void sem_wait (semaphore s) {
    s.count --;
    if (s.count < 0) {
        // place process in s.queue
        // block process
    }
}

void sem_post (semaphore s) {
    s.count ++;
    if (s.count <= 0) {
        // remove process P from s.queue
        // place P in the ready list
    }
}
```

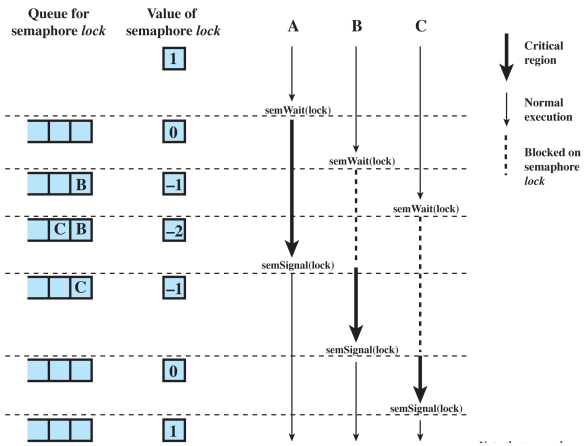

SEMAPHORES (CONT'D)

- A queue is used to hold the processes waiting for the semaphore
- Strong Semaphores: the process that has been blocked the longest is released from the queue first (FIFO)
- Weak Semaphores: the order in which processes are removed from the queue is not specified
- Mutual exclusion using semaphores:

```
semaphore s = 1;
void P(int i) {
    while (true) {
        // preceding code
        sem_wait(s);
        // critical section
        sem_post(s);
        // remainder
    }
}
void main() {
    // launch in parallel P(1), ..., P(n)
}
```

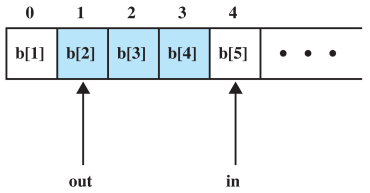


Semaphores Protecting Shared Data



The Producer/Consumer Problem

- General situation:
  - One or more producers are generating data and placing them in a buffer
  - A single consumer is taking items out of the buffer one at time
  - Only one producer or consumer may access the buffer at any one time
- The Problem: ensure that the producer cannot add data into a full buffer and the consumer cannot remove data from an empty buffer
- First variant: infinite buffer



The Producer/Consumer Problem: A First Solution

```
int n;
binary_semaphore s = 1, delay = 0;

void producer () {
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}

void consumer () {
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}

void main () {
     n = 0;
     // run in parallel producer(), consumer()
}
```

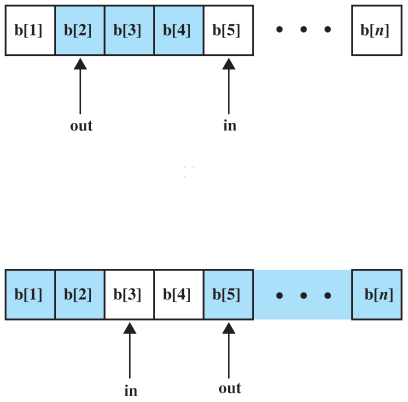

The Producer/Consumer Problem: A First (Incorrect) Solution

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if (n==1) {semSignalB(delay)}		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n–	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	if (n==1) {semSignalB(delay)}		0	1	1
13	semSignalB(s)		1	1	1
14		if (n==0) {semWaitB(delay)}	1	1	1
15		semWaitB(s)	0	1	1
16		n–	0	0	1
17		semSignalB(s)	1	0	1
18		if (n==0) {semWaitB(delay)}	1	0	0
19		semWaitB(s)	0	0	0
20		n–	0	-1	0
21		semSignalB(s)	1	-1	0

THE PRODUCER/CONSUMER PROBLEM: A CORRECT SOLUTION

```
int n;
binary_semaphore s = 1, delay = 0;

void producer () {
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}

void consumer () {
     int m; /* a local variable */
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}

void main () {
     n = 0;
     // run in parallel producer(), consumer()
}
```


PRODUCER/CONSUMER WITH FINITE CIRCULAR BUFFER



PRODUCER/CONSUMER WITH FINITE CIRCULAR BUFFER (CONT'D)

```
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;

void producer () {
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}

void consumer () {
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}

void main () {
     // run in parallel producer(), consumer()
}
```


SEMAPHORE IMPLEMENTATION

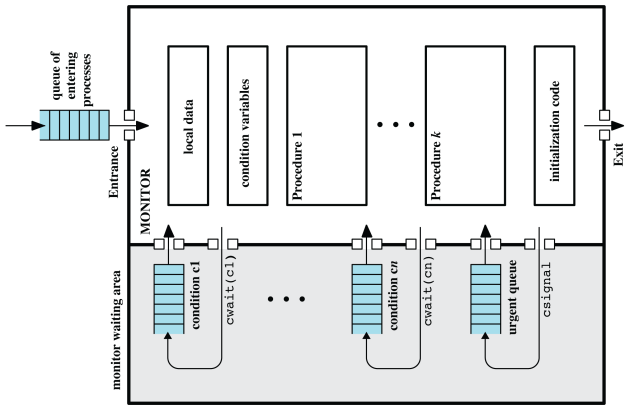
- Imperative that the semWait and semSignal operations be implemented as atomic primitives
- Can be implemented in hardware or firmware
- Best solution is to use one of the hardware-supported schemes for mutual exclusion
- Software schemes also exist
  - Peterson's algorithm: two process solution
  - Assume LOAD and STORE are atomic instructions (cannot be interrupted)
  - The two processes share two variables: `int turn; boolean flag[2];`
  - The variable `turn` indicates whose turn it is to enter the critical section.
  - The `flag` array is used to indicate if a process is ready to enter the critical section.
  - `flag[i] == true` implies that process P_i is ready!

```
void P (int i) {
  repeat {
    flag[i] = TRUE;
    j = i % 2;
    turn = j;
    while (flag[j] &&
           turn == j) ;
    // critical section
    flag[i] = FALSE;
    // remaining code
  }
}
```

MONITORS

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented as a library or directly into a number of programming languages
  - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java, etc.
- Software module consisting of one or more procedures, an initialization sequence, and local data
- Semantics:
  - Local data variables are accessible only by the monitor's procedures and not by any external procedure
  - Process enters monitor by invoking one of its procedures
  - Only one process may be executing in the monitor at a time
- Synchronization: Use of condition variables operated upon by two functions:
  - cwait(c): suspend execution of the calling process on condition c
  - csignal(c): resume execution of some process blocked on cwait on the same condition

MONITORS (CONT'D)



PRODUCER/CONSUMER SOLUTION USING A MONITOR

```
monitor boundedbuffer {
  char buffer [N];
  int nextin, nextout;
  int count;
  cond notfull, notempty;
  void append (char x) {
     if (count == N) cwait(notfull);
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);
  }
  void take (char x) {
     if (count == 0) cwait(notempty);
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;
     csignal(notfull);
  }
  {
     nextin = 0; nextout = 0; count = 0;
  }
}
```

```
void producer () {
     char x;
     while (true) {
     produce(x);
     append(x);
     }
}
void consumer () {
     char x;
     while (true) {
       take(x);
       consume(x);
     }
}

void main () {
     // run in parallel
     // producer(), consumer()
}
```


MESSAGE PASSING

- When processes interact with one another two fundamental requirements must be satisfied: synchronization (to enforce mutual exclusion) and communication (to exchange information)
- Message passing is one approach to providing both of these functions
  - Works with distributed systems as well as shared memory multiprocessor and uniprocessor systems
- Function normally provided in the form of a pair of primitives: send(destination, message) and receive(source, message)
  - A process sends information in the form of a message to another process designated by a destination
  - A process receives information by executing the receive primitive, indicating the source and the message

Message Passing Design Decisions

- Synchronization
  - Send: blocking or nonblocking
  - Receive: blocking, nonblocking, test for arrival
- Addressing
  - Direct: send, receive (explicit, implicit)
  - Indirect: static, dynamic, ownership
- Format: content, length (fixed, variable)
- Queuing discipline: FIFO, priority

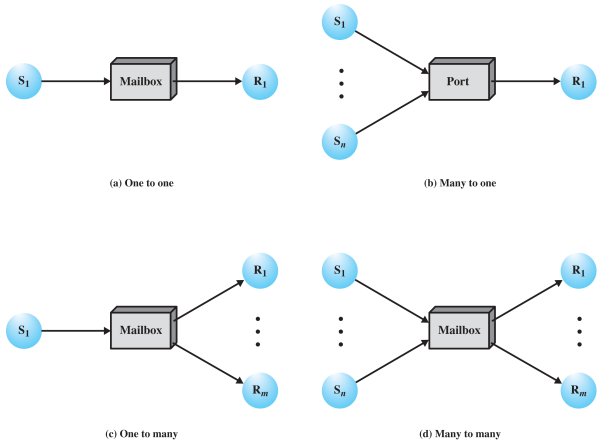
Synchronization in Message Passing

- Communication implies synchronization (receiver cannot receive the message until it has been sent)
- Rendevous: both sender and receiver blocked until message is delivered = tight synchronization
- Nonblocking send, blocking receive: most useful combination
  - Sender continues on but receiver is blocked until the requested message arrives
  - Sends one or more messages to a variety of destinations as quickly as possible
  - Example: a service process that exists to provide a service or resource to other processes
- Nonblocking send, nonblocking receive: neither party is required to wait

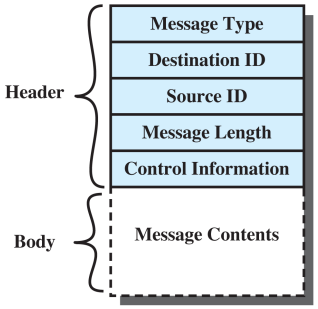
Addressing in Message Passing

- Direct addressing
  - Send includes a specific identifier of the destination process
  - Receive primitive can be handled in one of two ways:
    - Explicit: require that the process explicitly designate a sending process – effective for cooperating concurrent processes
    - Implicit: source parameter of the receive primitive possesses a value returned when the receive operation has been performed
- Indirect addressing
  - Messages are sent to a shared data structure consisting of queues that can temporarily hold messages
  - Queues are referred to as mailboxes
  - One process sends a message to the mailbox and the other process picks up the message from the mailbox
  - Allows for greater flexibility in the use of messages

Mailboxes



General Message Format



Message Passing Example: Unix Pipes

- One of the oldest and most elegant communication system is the pipe
  - Nonblocking send, blocking receive
  - Indirect addressing
  - Fully buffered stream paradigm (flow control)
- The system call pipe produces two file descriptors

```
char com[2];
r = pipe (com);
```

- Everything written into `com[1]` can be read back from `com[0]` in the same order

Mutual Exclusion Using Messages

```
void P (int i) {
    message msg;
    while (true) {
      receive (box, msg);
      /* critical section  */;
      send (box, msg);
      /* remainder   */;
    }
}

void main() {
    create_mailbox (box);
    send (box, null);
    // run in parallel P(1), P(2), ..., P(n)
}
```


Producer/Consumer Solution Using Message Passing

```
const int capacity = /* buffering capacity */ ;
          null = /* empty message */ ;
int i;

void producer () {
    message pmsg;
    while (true) {
      receive (mayproduce, pmsg);
      pmsg = produce();
      send (mayconsume, pmsg);
    }
}

void consumer () {
    message cmsg;
    while (true) {
      receive (mayconsume, cmsg);
      consume (cmsg);
      send (mayproduce, null);
    }
}

void main () {
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce, null);
    // run producer(), consumer()
}
```

The Readers/Writers Problem

- A data area (file) is shared among many processes
- Some processes only read the data area (readers) and some only write to the data area (writers)
- Conditions that must be satisfied:
  1. Any number of readers may simultaneously read the file
  2. Only one writer at a time may write to the file
  3. If a writer is writing to the file, no reader may read it
- Either readers or writers can have priority

Readers Have Priority

```
int readcount;
semaphore x = 1, wsem = 1;

void reader () {
    while (true) {
      semWait (x);
      readcount++;
      if (readcount == 1) semWait (wsem);
      semSignal (x);
      READUNIT();
      semWait (x);
      readcount--;
      if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
  }

void writer () {
    while (true) {
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
    }
}

void main() {
    readcount = 0;
    // run reader(), writer()
}
```


Writers Have Priority

```
int readcount, writecount;  semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;

void reader () {
    while (true) {
      semWait (z);
       semWait (rsem);
        semWait (x);
          readcount++;
          if (readcount == 1)
              semWait (wsem);
       semSignal (x);
       semSignal (rsem);
      semSignal (z);
      READUNIT();
      semWait (x);
       readcount--;
       if (readcount == 0)
           semSignal (wsem);
      semSignal (x);
    }
}
void main() {  readcount = writecount = 0;
               // run reader(), writer()     }
```

```
void writer () {
    while (true) {
      semWait (y);
       writecount++;
       if (writecount == 1)
           semWait (rsem);
      semSignal (y);
      semWait (wsem);
       WRITEUNIT();
      semSignal (wsem);
      semWait (y);
       writecount--;
       if (writecount == 0)
           semSignal (rsem);
      semSignal (y);
    }
}
```


Writers Have Priority (cont'd)

- Readers only in the system: wsem set, no queues
- Writers only in the system: wsem and rsem set, writers queue on wsem
- Both readers and writers with read first
  - wsem set by reader
  - rsem set by writer
  - all writers queue on wsem
  - one reader queues on rsem
  - other readers queue on z
- Both readers and writers with write first
  - wsem set by writer
  - rsem set by writer
  - writers queue on wsem
  - one reader queues on rsem
  - other readers queue on z

Readers/Writers with Message Passing

```
void reader (int i) {
    message rmsg;
       while (true) {
          rmsg = i;
          send (readrequest, rmsg);
          receive (mbox[i], rmsg);
          READUNIT ();
          rmsg = i;
          send (finished, rmsg);
       }
}

void writer (int j) {
    message rmsg;
    while(true) {
       rmsg = j;
       send (writerequest, rmsg);
       receive (mbox[j], rmsg);
       WRITEUNIT ();
       rmsg = j;
       send (finished, rmsg);
    }
}
```

```
void controller () {
     while (true) {
        if (count > 0) {
           if (!empty (finished)) {
              receive (finished, msg);
              count++;            }
           else if (!empty (writerequest)) {
              receive (writerequest, msg);
              writer_id = msg.id;
              count = count - 100;          }
           else if (!empty (readrequest)) {
              receive (readrequest, msg);
              count--;
              send (msg.id, "OK");        }
        }
        if (count == 0) {
           send (writer_id, "OK");
           receive (finished, msg);
           count = 100; }
        while (count < 0) {
           receive (finished, msg);
           count++;       }
     }  }
```


File Locking

Scenario:

- We have a concurrent network server that accesses a file f
- A server process wants to write a continuous block of data into f
- It is quite possible that two processes write to f at the same time, ending up with corrupted data

Solution: A process that wants to write to the file acquires first a (POSIX) lock: With a file descriptor `fd`, we do: `lockf(fd,F_LOCK,0);`

- If a file is already locked, `lockf(fd,F_LOCK,0)` blocks (or returns failure, see the `O_NONBLOCK` flag of `open`)
- It might also be a good idea to see whether a file is locked: `lockf(fd,F_TEST,0);` (returns -1 and `errno` is set to `EAGAIN`)
- If a file is locked by other process, `write` will block until the lock is released (beware of deadlocks!)
- To unlock the file: `lockf(fd,F_ULOCK,0);`

File Locking (cont'd)

- The lock created by `lockf` is mandatory, and enforced by the kernel
  - The lock is however only for writing; a process can still read from a locked file
  - You can lock portions of files, by giving a non-zero third argument (the offset) to `lockf`
- To obtain more flexible locks, you should use `fcntl`

```
struct flock lock;
fcntl(fd,F_GETLK,&lock);  // get the lock status in l_type (below), or
fcntl(fd,F_SETLK,&lock);  // tries to lock, returns -1 on insuccess, or
fcntl(fd,F_SETLKW,&lock); // tries to lock, wait as long as necessary
                          // (deadlock possible!)
```

  - The `flock` structure has the following fields:

`l_type`	type of lock: `F_RDLCK`, `F_WRLCK`, or `F_UNLCK` (set by `F_GETLK`)
`l_whence`	where `l_start` is relative to (like third argument of `lseek`)
`l_start`	offset where the lock begins
`l_len`	size of the locked area (zero means until EOF)
`l_pid`	process holding the lock

File Locking Implementation

- The theory is nice by the practice in messy...
- On Linux `lockf(3)` is just a wrapper for `fcntl(2)`; on other systems this might not be the case (so don't mix and match)
- On BSD mandatory and advisory locking are aware of each other (so you can mix and match at will), other systems might see things differently
- A lock is mandatory iff the file is marked as a candidate for mandatory locking by setting the group-id bit in its file mode but removing the group-execute bit — an otherwise meaningless combination chosen so as not to break existing user programs
- Solaris, HP-UX reject all calls to `open()` for a file on which another process has outstanding mandatory locks — direct contravention of the System V standard (but not POSIX), which states that only calls to `open()` with the `O_TRUNC` flag set should be rejected; Linux follow the System V standard
- Race conditions can still appear
- Quote from the Linux kernel: I'm afraid that this is such an esoteric area that the semantics described below are just as valid as any others, so long as the main points seem to agree

File locking Implementation (Linux)

- Mandatory locks can only be applied via the `fcntl()`/`lockf()` locking interface = the System V/POSIX interface. BSD style locks using `flock()` never result in a mandatory lock
- If a process has locked a region of a file with a mandatory read lock, then other processes are permitted to read from that region. If any of these processes attempts to write to the region it will block until the lock is released, unless the process has opened the file with the `O_NONBLOCK` flag in which case the system call will return immediately with the error status `EAGAIN`
- If a process has locked a region of a file with a mandatory write lock, all attempts to read or write to that region block until the lock is released, unless a process has opened the file with the `O_NONBLOCK` flag in which case the system call will return immediately with the error status `EAGAIN`
- Calls to `open()` with `O_TRUNC`, or to `creat()`, on a existing file that has any mandatory locks owned by other processes will be rejected with the error status `EAGAIN`
- Not even root can override a mandatory lock, so runaway processes can wreak havoc if they lock crucial files!
  - Way around: remove the setgid bit (hard to do on a hung system)

The perils of POSIX locks

- POSIX locks (`lockf`) are mandatory and enforced by the kernel
- . . . but there are times in which they do not do what you want them to do:
  - Contrary to the popular belief, the following program will not block on the lock:

```
int fd = open("aaa", O_RDWR);
if (fork() == 0) {
    lockf(fd,F_LOCK,0);
    printf("child locked \"aaa\"..."); sleep(5); printf(" done\n");
}
else {
    sleep(1);
    write(fd,"stuff",5);
    printf("parent wrote into \"aaa\"\n");
}
```

  - POSIX locks are established at descriptor level and are thus inherited by child processes
  - However, attempting to place a lock on an already locked descriptor will positively block

Advisory locks

- Advisory locks (`flock`) look almost the same. . . but are only advisory
- That is, any program may choose not to take them into account
- Avoiding `flock` in favour of POSIX locks is probably a good idea
  - but then do take into consideration the issues related to your program locking a crucial file and then hanging
- Advisory locks are by the way inherited by a child process but not across `execve`