The Process in unix like OS

Inter Process Communication

Apurba Sarkar

Department of Computer Science and Tech.
Bengal Engg. and Sc. University Shibpur.

August 29, 2012

fork() again

- The only way in which a new process is created by Unix system is by executing the `fork()` system call.
- the `fork()` system call creates a copy of the the process that was executing.
- The process that calls the `fork()` is called the *parent process* and the new process is called the *child process*.
- Realize that if the text segment can be shared, then both the parent and child can share the text segment after `fork()`.

fork() again

- Realize that the child's copy of the data segment is the copy of the parent's data segment when the `fork()` operation takes place. It is not a copy from the program's disk file.
- The files that were open in the parent process before the `fork()` are shared by the child process after the `fork()`.
- This provides an easy way for the parent process to open specific files or devices and pass these open files to the child process.
- After the `fork()` parent closes the files that it opened for the child, so that both the processes are not sharing the same file.

fork() again

- Regarding the process variable that we were discussing in last couple of classes, the values of the following in the child process are copied from the parent
  - `real user ID`
  - `real group ID`
  - `effective user ID`
  - `effective group ID`
  - `process group ID`
  - `root directory`
  - `current working directory`
  - `signal handling settings`
  - `file mode creation mask`

fork() again

- The child process differs from the parent process in the following ways
    - the child process has a new, unique process ID
    - the child proces has a diffrent parent process ID
    - the child process has its own copies of the parent's file descriptors
    - the time left untill an alarm clock signal is set zero in the child

uses of fork()

- There are two uses of `fork()`
  1. A process wants to make a copy of itself so that one copy can handle an operation while the other copy does another task. This is typical for network servers.
  2. A process wants to execute another program. Since the only way to create a new process is with the `fork()` operation, the process must `fork()` to make a copy of itself, then one copy issues an `exec()` to execute the new program. this is typical for programs such as shells.

feature of exec()

- The program invoked by `exec()` system call inherits the following attributes from the process that calls `exec()`
    - `real user ID`
    - `real group ID`
    - `process ID`
    - `parent process ID`
    - `process group ID`
    - `time left untill am alarm clock signal`
    - `root directory`
    - `current working directory`
    - `file mode creation mask`
    - `file locks`

feature of exec()

- two attributes that can change when a new program is execed are
  - `effective user ID`
  - `effective group ID`
- if the set-user-id bit is set for the program being execed, the effective user ID is changed to the user ID of the owner of the program file.
- similarly, if the ser-group-id bit is set for the program being execed, the effective group ID is changed to the group ID of the owner of the program file.

dup() and dup2() system call

dup() An exixting file descriptor is duplicated by

`int dup(int fildes);`

- This returns a new file descriptor that shares the following with original *fildes*:
  - both refers to the same file or pipe
  - the access mode of the new file descriptor is the same as that of the original: read, write, or read/write.
  - both the file descriptors share the same file position.
  - it is guranteed that the new file descriptor returned by `dup()` is the lowest numbered available file descriptor.

dup() and dup2() system call

dup2() 4.3BSD provides another variant of this system call

```
int dup2(int oldfildes,int newfildes);
```

- here the *newfildes* argument specifies the new descriptor value that is desired. If this new descriptor value is already in use, it is first released, as if a `close()` had been done.

fork() and file sharing

- The sharing of files across a `fork()` system call merits further review, as it nicely describe the sharing of files between unix processes.
- There are three kernel data structures(actly tables) used to access a file
  - Every process has a process table entry. one portion of the process table is an array of file pointers. the file descriptors are just indexes into this array.
  - The file pointers in the process table point to entries in the file table. The file table is where the current file position for all open files in the system is maintained.
  - Another table maintained by kernel is *`i-node table`*. Every open file has an entry in the *`i-node table`*. An inode table entry contains all the inforamtion read from i-node on disk along with some other fields that the kernel needs to maintain.

fork() and file sharing

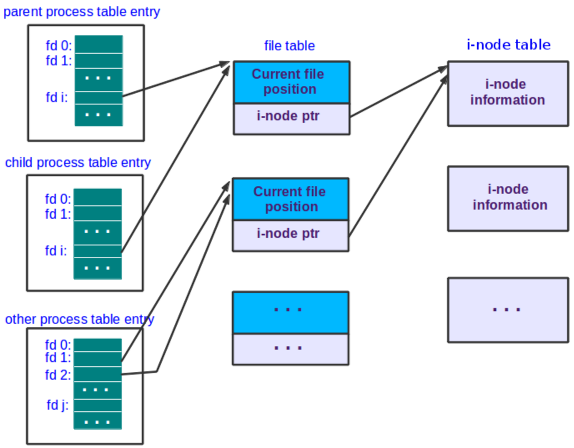

Figure 1: The sharing of files between processes

What is pipe?

- A pipe is used for one-way communication of a stream of bytes.
- The command to create a pipe is `pipe()`, which takes an array of two integers.
- It fills in the array with two file descriptors that can be used for low-level I/O. The following code shows how to crate a pipe,

```
int pfd[2];
pipe(pfd);
```

What is pipe? contd

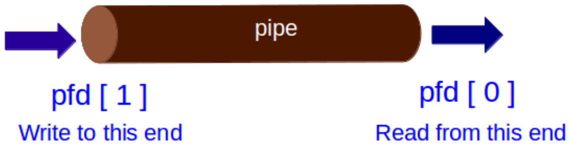

Figure 2: A pipe

I/O with pipe

- File descriptors created by pipe system call can be used for block I/O.
- Following code snippet shows how to write to and read from a pipe. `write(pfd[1], buf, SIZE);`
  `read(pfd[0], buf, SIZE);`

pipe and fork

- A single process would not use a pipe.
- They are used when two processes wish to communicate in a one-way fashion.
- A process splits in two using fork().
- A pipe opened before the fork becomes shared between the two processes.

I/O with pipe

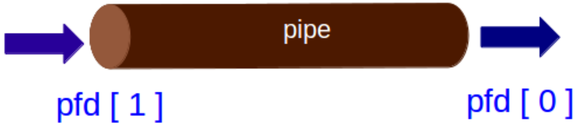

Figure 3: pipe before fork()

pipe and fork contd

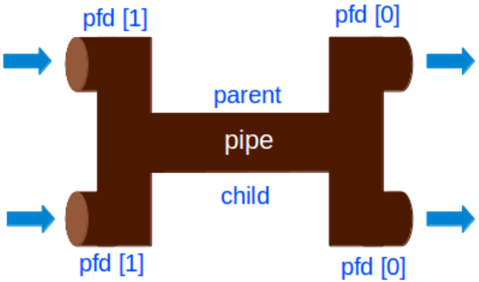

Figure 4: pipe after fork()

pipe and fork contd

- This gives two read ends and two write ends.
- The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.
- Either process can write into the pipe, and either can read from it.Which process will get what is not known.
- For predictable behaviour, one of the processes must close its read end, and the other must close its write end to make it a simple pipeline again.

pipe and fork contd

- Suppose the parent wants to write down a pipeline to a child.
- The parent closes its read end, and writes into the other end.
- The child closes its write end and reads from the other end.
- When the processes have ceased communication, the parent closes its write end.
- This means that the child gets eof on its next read, and it can close its read end. This is shown in the next slide.

pipe and fork contd

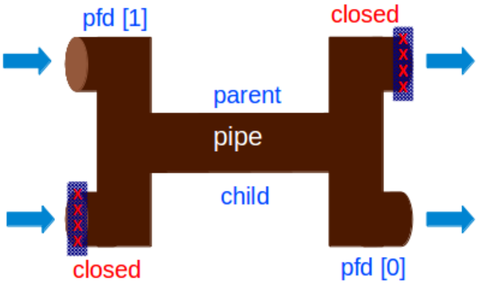

Figure 5: simple pipe line

An example

```
#include <stdio.h>
#define SIZE 1024
int main(int argc, char **argv)
{
  int pfd[2],nread,pid;
  char buf[SIZE];
  if (pipe(pfd) == -1){
  perror("pipe failed");
    exit(1);
  }
  if ((pid = fork()) < 0){
    perror("fork failed");
    exit(2);
  }
```

An example

```
    if (pid == 0) /* child */
    {close(pfd[1]);
      while ((nread = read(pfd[0], buf, SIZE))!= 0)
        printf("child read %s\n", buf);
        close(pfd[0]);
    } else {
      close(pfd[0]); /* parent */
      strcpy(buf, "hello...");
      /* include null terminator in write */
      write(pfd[1], buf,strlen(buf)+1);
      close(pfd[1]);
    }
    exit(0);
  }
```

dup system call

- A pipeline works because the two processes know the file descriptor of each end of the pipe.
- Each process has a stdin (0), a stdout (1) and a stderr (2).
- The file descriptors will depend on which other files have been opened, but could be 3 and 4, say.
- If one of the processes replaces itself by an “exec”. The new process will have files for descriptors 0, 1, 2, 3 and 4 open.

How will it know which are the ones belonging to the pipe?

pipe A real example “ls|wc”

- To implement “ls | wc” the shell will have created a pipe and then forked.
- The parent will exec to be replaced by “ls”, and the child will exec to be replaced by “wc”
- The write end of the pipe may be descriptor 3 and the read end may be descriptor 4.
- “ls” normally writes to 1 and “wc” normally reads from 0.

pipe A real example “ls|wc”

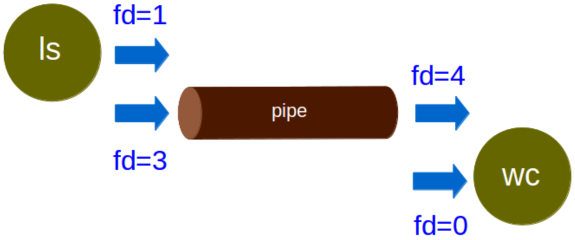

Figure 6: Before dup system call

pipe A real example "ls|wc"

- The dup2() system call takes an existing file descriptor, and another one that it "would like to be".
- Here, fd=3 would also like to be 1, and fd=4 would like to be 0.
- So we dup2 fd=3 as 1, and dup2 fd=4 as 0. Then the old fd=3 and fd=4 can be closed as they are no longer needed.

pipe A real example "ls|wc"

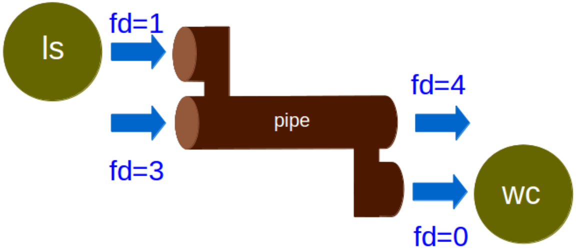

Figure 7: After dup system call

pipe A real example “ls|wc”

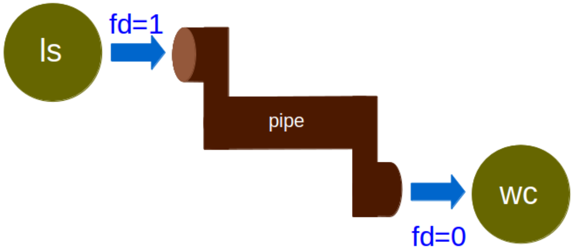

Figure 8: After close system call

pipe and fork example

Without any error checks, the program to do this is

```
int main(void)
{ int pfd[2],pipe(pfd);
  if (fork() == 0) {
    close(pfd[1]);
    dup2(pfd[0], 0);
    close(pfd[0]);
    execlp("wc", "wc",(char *) 0);
  } else {
    close(pfd[0]);
    dup2(pfd[1], 1);
    close(pfd[1]);
    execlp("ls", "ls",(char *) 0);
  }
}
```

Summery

- pipe system call is always followed by a fork system call.
- As an effect of the `fork` the ends of the pipe are shared.
- We generally close one read end and one write end of the pipe to create a one way channel.
- pipe can be used for one way communication between two processes only when they have parent-child relationship.
- A pipeline may consist of three or more process (such as a C version of `ps | sed 1d | wc -l`).
- A process may want to both write to and read from a child. In this case it creates two pipes.

Why Share memory?

Why Share memory?

- pipe is used for communication between two related process.

Why Share memory?

- pipe is used for communication between two related process.
- What if the they are not related and still want to communicate?

Why Share memory?

- pipe is used for communication between two related process.
- What if the they are not related and still want to communicate?
- shared memory comes handy in such situation.

Systeam calls needed

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

int shmget(key_t key, int size, int shmflg);

char *shmat(int shmid, char *shmaddr, int shmflg);

int shmdt(char *shmaddr);
```

shmget()

- It creates or locates a piece of shared memory.
- The pieces are identified by a key.
- The key is an integer agreed upon by all the communicating processes.
- It also specifies the size of the memory and the permission flags (eg -rwx——).

shmat()

- It takes the shared memory identifier and maps that into the address space of the process.
- Returns a pointer to it.
- The key is an integer agreed upon by all the communicating processes.
- The pointer then on can be used just like any pointer to memory .

shmat()

- It takes the shared memory identifier and maps that into the address space of the process.
- Returns a pointer to it.
- The key is an integer agreed upon by all the communicating processes.
- The pointer then on can be used just like any pointer to memory .

Critical section problem

- Suppose there are n processes $P_0, P_1, \ldots, P_{n-1}$
- Each process has a segment of code, called critical section, in which the process may be changing common variables, updating a table, writing a file, and so on.
- Each process must request permission to enter its critical section.
- The section of code implementing this request is *entry* section and
- the remaining code is *remainder* section.

Critical section problem

General structure of a process P_i

```
repeat

    $\boxed{\textit{entry section}}$

        critical section

    $\boxed{\textit{exit section}}$

        remainder section

until $\mathit{false}$;
```

Critical section problem

Solution to csp must satisfy the following three requirements

1. **Mutual Exclusion:** If process P_i is executing in its critical section, then no other process can be executing in their critical section.
2. **Progress:** If no process is executing in its CS and there exist some processes that wish to enter their CS, then the selection of the process that will enter the CS next cannot be postponed indefinitely.
3. **Bounded Waiting:** There exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Assumptions

Solutions to Critical Section Problems are based on the following assumptions.

1. No assumptions made about relative speed of processes
2. No process may remain in its critical section indefinitely (may not terminate in its critical section)
3. A memory operation (read or write) is atomic — cannot be interrupted.

Algorithm 1

- Software solutions
  - Algorithms whose correctness does not rely on any assumptions other than positive processing speed (that may mean no failure).
  - Busy waiting.
- Hardware solutions
  - Rely on some special machine instructions.
- Operating system solutions
  - Extending hardware solutions to provide some functions and data structure support to the programmer.

Solution to csp Algorithm 1

General structure of a process P_i

```
repeat
    while $turn \neq i$ do $no\text{-}op$
        critical section
    $turn = j$
        remainder section
until $false$;
```

Algorithm 1

- We let the Processes to share a common integer variable *turn* initialized to 0 or 1.
- if *turn = i*, then process P_i is allowed to execute in its critical section.
- This solution ensures that only one process at a time can be in its critical section.
- However, it does not satisfy the progress requirement, since it requires strict alternation of process in the execution of the critical section.
- if *turn=0*, and P_1 is ready to enter its critical section, P_1 can not do so, even though P_0 may be in its remainder section.

Algorithm 2

- The problem with algorithm1 is that it does not retain sufficient info abt the state of each process.
- it remembers only which process is allowed to enter that process critical section.
- To remedy this, we can replace the variable *turn* with the following array.
  var *flag:* array[0 . . .1] of *boolean*;
- The elements of the array are initialized to *false*.
- If *flag[i]=true*, this value indicates that P_i is ready to enter critical section.

Solution to csp Algorithm 2

General structure of a process P_i

```
repeat

    flag[i] := true;
    while flag[j] do no-op;

        critical section

    flag[i] := false;

        remainder section

until false;
```

Algorithm2

- In this algorithm, process P_i first sets `flag[i]` to be `true`, signalling that it is ready to enter critical section.
- Then, P_i checks whether P_j is ready to enter critical section or not.
- If P_j were ready, then P_i would wait until P_j had indicated that it is done with the critical section.(setting `flag[j]=false`)
- P_i would enter and on exiting the CS it would set `flag[i]` to be `false`, allowing other to enter its CS.

Problems with Algorithm2

- Mutual exclusion requirement is satisfied.
- Unfortunately progress requirement is not satisfied.
  Lets verify ...
  consider the following execution sequence:

  T_0: P_0 sets `flag[0] = true`
  T_1: P_1 sets `flag[1] = true`

- P_0 and P_1 are looping forever in their respective **while** statements.

Problems with Algorithm2

- This algorithm is crucially dependent on the exact timing of the two processes.
- This sequence could have been derived in an environment where there are multiple processors executing concurrently, or
- where an interrupt occurs immediately after step T_0 is executed, and the CPU is switched from one process to another.
- Note that **switching the order** of the instructions will not solve the problem. Rather we will arrive at a situation where both the processes will be at the CS at the same time.

Algorithm3

- Let us combine key ideas of algorithm 1 and algorithm 2 and obtain a new algorithm.
- We notice this solution is correct and all three requirements ar satisfied,
- Now processes share two variables:
  var *flag:* array[0 . . .1] of *boolean*;
  turn : 0..1;
- initially *flag[0]=flag[1]=false* and the value of *turn* is immaterial.

Solution to CS Algorithm 3

General structure of a process P_i

```
repeat
    flag[i] := true;
    turn := j;
    while (flag[j] and turn = j) do no-op;
        critical section
    flag[i] := false;
        remainder section
until false;
```

Algorithm3

- To enter the critical section, process P_i first sets `flag[i]` to be `true`,
- Asserts that it is the other process's turn to enter if appropriate `turn = j`.
- if both processes try to enter at the same time, turn will be set to both *i* and *j* at roughly the same time.Only one of these value will last; the other will occur, but will be overwritten immediately.
- the eventual value of `turn` decides which of the two process is allowed to enter its critical section first.

Algorithm3 claims

1. Mutual exclusion is preserved,
2. The progress requirement is satisfied,
3. The bounded-waiting requirement is met.

Algorithm3 proof

1. To prove property 1, we note that P_i enter its critical section only if either `flag[j] = false` or `turn = i`. Also note that, if both processes can be executing in their critical sections at the same tie, then `flag[0]=flag[1]=true`. These two observation imply that P_0 and P_1 could not have executed successfully their **while** statements at about the same time, since the value of `turn` can be either 0 or 1 but cannot be both.
2. Hence, one of the process, say P_j must have executed successfully the **while** statement, whereas P_i had to execute at least one additional statement ("`turn = j`"). However, since, at that time `flag[j] = true`, and `turn = j`, and this condition persists as long as P_j is in its critical section, the result follows.

Multi-process solution

1. This algorithm is known as Bakery algorithm and based on a scheduling algorithm commomnly used in bakeries, ice-cream stores, meat markets etc.
2. Generalization for n processes.
3. Before entering its critical section, a process receives a number. The holder of the smallest number enters its critical section.
4. Tie breaker is done using the process id: if processes i and j hold the same number and $i < j$ then i enters first.

Bakery Algorithm

1. The common data structure are
   var *flag:* `array[0 ...1] of` *boolean*;
   var *label:* `array[0 ...1] of` *integer*

   Initially these data structures are initialized to false and 0, respectively,

2. We define following notation for convenience.
   - $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$.
   - *max*$(a_0, \ldots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for $i = 0, \ldots n - 1$.

Structure of P_i in Bakery Algorithm

```
repeat
    flag[i] := true;
    label[i] := max(label[0], label[1],..., label[n-1])+1;
    while((∃k ≠ i)(flag[k] && (label[k],k) <(label[i],i))) do no op;

        critical section

    flag[i]=false;

        remainder section

until false;
```

An aside

★ Put suitable code in place of **condition** so that both the string “Hello” and “world” will be printed on the standard output

Condition ???

```
#include<stdio.h>
int main()
{
    if(condition)
    {
      printf("Hello\n");
    }
    else
    {
      printf("world\n");
    }

}
```

Proofs

Lemma

The Bakery lock algorithm is deadlock free.

Proof.

Some waiting thread A has the unique least `(label[A],A)` pair, and that thread never waits for another thread. □

Proofs

Lemma

The Bakery lock algorithm is first-come-first-served.

Proof.

If A's doorway[a] preceeds B's, $D_A \rightarrow D_B$ then A's label is smaller since $write_A(label[A]) \rightarrow read_B(label[A]) \rightarrow write_B(label[B]) \rightarrow read_B(flag[A])$,

so B is locked out while `flag[A]` is `true` $\square$

[a]Doorway is the code `flag[i]=true;` and `label[i]=max(...);`

★ Note that any algorithm that is both `deadlock-free` and `first-come-first-served` is also `starvation-free`

Classic example of Deadlock

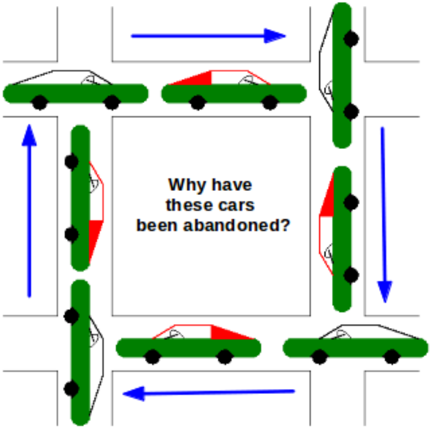

Figure 9: Classic car deadlock

Proofs

Lemma

The Bakery lock algorithm satisfies mutual exclusion

Proof.

Suppose not. Let A and B be two threads concurrently in the critical section. Let labeling$_A$ and labeling$_B$ be the last respective sequences of acquiring new labels prior to entering the critical section. Suppose that `(label[A],A)`$\ll$`(label[B],B)`. When B successfully completed the test in its waiting section, it must have read that `flag[A]` was $\mathtt{false}$ or that that `(label[B],B)`$\ll$`(label[A],A)`. However, for a given process, its $\mathtt{id}$ is fixed and `label[]` values are strictly increasing, so B must have seen that `flag[A]` was $\mathtt{false}$.
it follows that $\mathtt{labeling}_B \rightarrow \mathtt{read}_B\mathtt{(flag[A])} \rightarrow \mathtt{write}_A\mathtt{(flag[A])} \rightarrow \mathtt{labeling}_A$.

Which contradicts the assumption that `(label[A],A)`$\ll$`(label[B],B)`. □

Hardware Solutions

- We could solve critical section problem in uniprocessor environment by simply disallowing interrupts to occur while shared variable is being modified.
- This way we could be sure that current sequence of instructions would be allowed to execute in order without preemption.
- Unfortunately this solution is not feasible in multiprocessor environment.
- Many machine therefore provide special hardware instruction that allow us either to test and modify content of word, or to swap the contents of two words, atomically.

Disabling Interrupt

General structure of a process P_i

```
repeat
    **Disable Interrupts**
        critical section
    **Enable Interrupts**
        remainder section
until $false$;
```

Problems with this approach:

- Disabling interrupts for long periods of time.
- Making I/O calls during CS.
- Not feasible in multiprocessor environment.

Hardware Instruction

- Many machine provide special hardware instruction that allow us either to test and modify the content of the a word, or to swap the contents of two words, atomically.
- These instructions can be used to solve critical section problem in a relatively simple manner.
- lets discuss few of them.

Test-and-Set

- The `Test-and-Set` instruction is given below

```
function Test-and-Set(var target: boolean): boolean;
   begin
     Test-and-Set := target;
     target := true;
   end
```

- This instruction is executed atomically as one uninterruptible unit.
- Thus if two `Test-and-Set` instruction are executed simultaneously they will be executed sequentially in some arbitrary order.

Synchronization using Test-and-Set

- If the machine supports the `Test-and-Set` instruction, then we can implement mutual exclusion by declaring a Boolean variable *lock*, initialized to *false*.

Structure of P_i

```
repeat
    while Test-and-Set(lock) do no-op;
            critical section
    lock := false;
            remainder section
until false;
```

Swap

- The Swap instruction is given below

```
procedure Swap(var a,b: boolean;)
  var temp: boolean;
       begin
          temp := a;
            a := b;
            b := temp;
       end
```

- The Swap instruction operates on the contents of two words.
- Like Test-and-Set it is executed atomically as one uninterruptible unit.
- Thus if two Swap instruction are executed simultaneously they will be executed sequentially in some arbitrary order.

Synchronization using Swap

- If the machine supports the Swap instruction, then mutual exclusion can be provided as follows.
    - A global variable *lock* is declared and initialized to *false*.
    - In addition, each process also has a local boolean variable *key*.
    - The structure of a process P_i is given below.

Structure of P_i

```
repeat
    key := true;
    repeat
    Swap(lock,key);
    until key == false;

        critical section

    lock := false;

        remainder section

until false;
```

Operating System Solutions Semaphore

- Synchronization tool provided by the OS
- Can only be accessed via two *atomic* operations *wait* and *Signal*.
- The classical definitions of *wait* and *signal* are as follows.

```
wait(s): while S ≤ 0 do no-op;
S := S - 1;
```

and

```
signal(s): S := S + 1;
```

Operating System Solutions Semaphore

- Modification to the integer value of the semaphore in the *wait* and *signal* operation must be executed indivisibly.
- In the case of `wait(S)`, the testing of the integer value of S $(S \leq 0)$ and its possible modification `(S := S - 1)`, must also be executed without interruption.

Semaphore Usage

- we can use semaphore to deal with the *n* process critical-section problem.
- The *n* processes share a semaphore, *mutex*(standing for mutual exclusion).
- Each process is organized as bellow

Structure of P_i

```
repeat
    wait(mutex);
            critical section
    signal(mutex)
            remainder section
until false;
```

Semaphore Usage

- Consider the two concurrently running processes: P_1 with a statement S_1, and P_2 with a statement S_2. Suppose that we require that S_2 be executed only after S_1 has completed.

Semaphore Usage

- Consider the two concurrently running processes: P_1 with a statement S_1, and P_2 with a statement S_2. Suppose that we require that S_2 be executed only after S_1 has completed.
- We can implement this scheme readily by letting P_1 and P_2 share a common variable *synch*, initialized to 0, and ? ? ?

Semaphore Usage

- Consider the two concurrently running processes: P_1 with a statement S_1, and P_2 with a statement S_2. Suppose that we require that S_2 be executed only after S_1 has completed.
- We can implement this scheme readily by letting P_1 and P_2 share a common variable $synch$, initialized to 0, and ? ? ?
- You Guys solve it.

Implementation

- The main disadvantage of the mutual-exclusion solutions we have discussed so far is that they all require *busy waiting*.
- While a process is in critical section, any other process that tries to enter critical section must loop continuously in the entry code
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is sometimes called *spinlock*
- *spinlock* are useful in multiprocessor system.
- Advantage of *spinlock* is that no context switch is required when a process must wait on a lock.
- When locks are expected to be held for short times, *spinlocks* are useful.

Implementation

- To overcome the need for busy waiting, we can modify the definition of the *wait* and *signal* semaphore operations
- When a process executes the *wait* operation and finds that the semaphore value is not positive, it must wait.
- However, this time rather than busy waiting, the process blocks itself.
- The block operation places the process into a waiting queue associated with that semaphore, and its state changed to waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.

Implementation

- The process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal operation.
- The process is restarted by *wakeup* operation, which changes the process from the waiting state to the ready state.
- The process is then placed in the ready queue.
  To implement semaphore under this definition, we define semaphore as the following record

```
type semaphore = record
value : integer
L: list of process
end
```

Implementation

- Each semaphore has an integer value and a list of processes.
- When a process must wait on a semaphore, it is added to the list of processes.
- A *signal* operation removes one process from the list of waiting processes, and awakens that process.

Implementation

The semaphore operation under new definition now becomes

```
wait(S): S.value := S.value - 1;
if S.value < 0
then begin
add this process to S.L block;
end
```

and

```
signal(S): S.value := S.value + 1;
if S.value ≤ 0
then begin
remove a process P from S.L
wakeup(P);
end
```

Implementation

- The *block* operation suspends the process that invokes it.
- The *wakeup(P)* operation resumes the execution of a blocked process P.
- Note that, although under the classical definition of semaphores with busy waiting, the semaphore value is never negative, this implementation may have negative value.
- If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This is actually the result of the switching of the order of the decrement and the test in the implementation of the *wait* operation.

Implementation

- The list of waiting processes can be easily implemented by link field in each process control block(PCB).
- Each semaphore contains an integer value and a pointer to a list of PCBs
- To ensure bounded waiting one way would be maintain first-in, first-out queue, with semaphore containing both head and tail pointers to the queue.
- However the list may use any queuing strategy.

Implementation

- The critical aspect of semaphore is that they are executed atomically
- We must guaranty that no two processes can execute $wait$ and $signal$ operations on the same semaphore at the same time.

Deadlocks and Starvation

- The semaphore implementation that we have discussed may result in deadlock if not used properly.
- To illustrate this, we consider a system consisting of two processes, P_0 and P_1, each accessing two semaphore S and Q set to the value 1.

  Suppose P_0 executes `wait(S)`, and P_1 executes `wait(Q)`. When P_0 executes `wait(Q)`, it must wait untill P_1 executes `signal(Q)`. Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`. Since these signal operation cannot be executed, P_0 and P_1 are deadlocked.

Deadlocks and Starvation

- The semaphore implementation that we have discussed may result in deadlock if not used properly.
- To illustrate this, we consider a system consisting of two processes, P_0 and P_1, each accessing two semaphore S and Q set to the value 1.

  Suppose P_0 executes `wait(S)`, and P_1 executes `wait(Q)`. When P_0 executes `wait(Q)`, it must wait untill P_1 executes `signal(Q)`. Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`. Since these signal operation cannot be executed, P_0 and P_1 are deadlocked.

P_0	P_1
`wait(S)`	`wait(Q)`
`wait(Q)`	`wait(S)`
⋮	⋮
`signal(S)`	`signal(Q)`
`signal(Q)`	`signal(S)`

Deadlocks and Starvation

- We say that a set of process is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.
- Another problem related to deadlocks is *indefinite blocking* or *starvation*, a situation where process wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with semaphore **LIFO** order.

Classical problem of Synchronization

- There are number of different synchronization problems that are important mainly because they are example for a large class of `concurrency-control` problems.
- These problems are used for testing nearly every newly proposed synchronization scheme. To name a few,
  1. Bounded buffer producer consumer problem
  2. The Readers Writers Problem
  3. The Dining-Philosophers Problem

Bounded buffer producer consumer problem

- There is a shared buffer of size n, each buffer location contains an item. There are say M producer and N consumer running concurrently. If there is no empty location(buffer is full) in the buffer the producer must wait, on the other hand if the buffer is empty the consumer must wait. Synchronize these producer and consumer so that no race condition occurs.

Producer

```
repeat
  produce an Item
  ...
  wait(empty)
  wait(mutex)
  ...
  buffer[in]=Item
  in=(in + 1) mod n
  ...
  signal(mutex)
  signal(full)
until false;
```

Consumer

```
repeat
  produce an Item
  ...
  wait(full)
  wait(mutex)
  ...
  Item=buffer[out]
  out=(out + 1) mod n
  ...
  signal(mutex)
  signal(empty)
until false;
```

Producer Consumer problem a few words

- The *mutex* semaphore provides mutual exclusion for access to the buffer pool and is initialized to the value 1.
- The *empty full* semaphore count the number of empty and full buffers, respectively. The semaphore *empty* is initialized to the value `n`; the semaphore *full* is initialized to the value 0.

The Readers and Writers Problem

- A data object(such as file or record) to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update(that is to read and write) the shared object.
- We differentiate between these two types of process by referring to those processes that are interested in only reading as *readers*, and to the rest as *writers*
- If two *readers* access the shared data object simultaneously, no adverse effects will result. However, if a *writer* and some other process (either a *reader* or a *writer*) access the shared object simultaneously chaos may result.

The Readers and Writers Problem

- To ensure that these difficulties do not arise, we require that the *writers* have exclusive access to the shared object.
- The readers-writers problem has several variation, all involving priorities.
- The simplest one, referred to as the *first* readers-writers problem, requires that no readers will be kept waiting unless a writer has already obtained permission to use the shared object. In other words no readers should wait for other readers to finish simply because a writer is waiting.

The Readers and Writers Problem

In the solution to the *first readers-writers* problem, the common data structures are

```
var mutex, wrt :  semaphore
  readcount:  integer;
```

Writer

```
repeat
  ...
  wait(wrt)
  ...
  signal(wrt)
until false;    ...
```

Reader

```
repeat
  wait(mutex)
   readcount=readcount+1;
   if readcount==1 then wait(wrt)
  signal(mutex)
  ...
   reading is performed;
  ...
  wait(mutex)
   readcount=readcount-1;
   if readcount==0 then wait(wrt)
  signal(mutex)
until false;
```

Case Study: Semaphore in Unix

- Is another form of IPC but they are not used for exchanging large amounts of data, as are PIPEs, FIFOs, and message queues
- They are mainly intended to let multiple processes synchronize their operations.
- Our Main use of `semaphore` will be to sunchronize the access to shared memory segments.

Case Study: Semaphore in Unix

- Consider Semaphore as integer valued variable that is resource counter.
- The value of the variable at any point in time is the number of resource units available.
- If we have one resource, say a file that is shared, then the valid semaphore values are zero and one.
- Since our use of semaphore is to provide resource synchronization between different processes, the actual semaphore value must be stored in the **Kernel** as shown below.

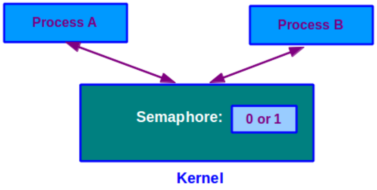

Figure 10: Semaphore value stored in Kernel