Process control under unix

Processes are created using the **fork**-system call. System call: the mechanism used by an application program to request service from the operating system (from the unix-kernel). **man -s2 intro**, **man -s2 syscalls**. **printf** (for example) is not a system call but a library function. **man -s3 intro** for details.

```
#include <sys/wait.h>   /* for wait            */
#include <sys/types.h>  /* for wait and fork   */
#include <unistd.h>     /* for fork and getppid */
#include <stdio.h>

int main()
{
  int             var, exit_stat;
  pid_t           pid;

  var = 10;
  printf("Before fork\n");

  if ((pid = fork()) < 0) {  /* note ( ) */
    printf("fork error\n");
    return 1;
  } else if (pid == 0) {     /* I am a child */
    var++;
    printf("child\n");
    sleep(60);               /* do some work  */
  } else {                   /* I am a parent */
    printf("parent\n");
    wait(&exit_stat);        /* wait for (one)    */
  }                          /* child to exit; not */
                             /* necessary to wait  */

  printf("ppid = %6ld, pid = %6ld, var  = %d\n",
         getppid(), pid, var); /* get parent proc id */
  return 0;
}
```



```
% a.out
Before fork
child
parent
ppid =   6843, pid =      0, var  = 11   child
ppid =  27174, pid =   6844, var  = 10   parent
```

fork creates the child process (from the parent process) by making a copy of the parent (so the child gets copies of the heap and stack for example). The child and parent continue executing with the instruction that follows the call to **fork**. So **fork** is called from the parent and returns both to the parent and the child.

Every process is identified by a number, the process id. or pid. We can list the pids (and some other properties) of all the processes running in the computer (this list has been shortened). The **ps**-commando takes a huge number of options.

```
% ps -fel | grep thomas
UID        PID    PPID   CMD
thomas    5442   27174   xterm
thomas    5446    5442   -csh

thomas    6843   27174   a.out  <-- parent
thomas    6844    6843   a.out  <-- child

thomas    6851    5446   ps -fel
thomas    6852    5446   grep thomas

thomas   27174   27171   -tcsh
thomas   27171   27152   sshd: thomas@pts/62
root     27152    3203   sshd: thomas [priv]
root      3203       1   /usr/sbin/sshd
root         1       0   init [5]
 ...
```


A process that hangs (not uncommon in parallel programming) can be terminated using the **kill**-command which sends a signal to a process. There are different signals and they can be used for communication between processes. Signal number 9, sigkill, cannot be caught.

```
% kill -l
HUP INT QUIT ILL TRAP ABRT BUS FPE KILL USR1 SEGV USR2
 ...

% ps U thomas
  PID TTY      STAT   TIME COMMAND
 8604 pts/62   S+     0:00 a.out   <-- kill this one
...
% kill -9 8604 (or kill -KILL 8604)
```

A process can choose to catch the signal using a a signal handler routine. It can also ignore (some) signals:

```
#include <signal.h>
#include <stdio.h>
int main()
{
  /* SIGINT is defined /usr/include/bits/signum.h */
  if ( sigignore(SIGINT) == -1 )
    printf("*** Error when calling sigignore.\n");

  while(1)      /* loop forever */
         ;
  return 0;
}

% gcc signal.c
% a.out
^C^C^C^C^C^C^C^C^C^\Quit

% /bin/stty -a
intr = ^C; quit = ^\; erase = ^H; etc....
```


To start a child process that differs from the parent we use the **exec** system call (there are several forms). **exec** replaces the child (the process which it is called from) with a new program.

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

int main()
{
  int             exit_stat;
  pid_t           pid;

  if ((pid = fork()) < 0) {
    printf("fork error\n");
    return 1;
  } else if (pid == 0) {  /* I am a child */
    /* replace this process by another  */
    /* execlp( file, name_shown_by_ps,
               arg1, ..., argn, NULL)   */

    /* (char *) 0 is a null pointer. (char*)
       is a type cast. See the C FAQ for details.*/

    /* new is a compiled C-program*/
    if(execlp("new", "new_name", (char*) 0) < 0) {
            printf("*** execlp error\n");
            return 1;
    }

  } else                  /* I am a parent. Wait */
    wait(&exit_stat);     /* or do something else */
  return 0;
}
```

Very common usage in **command&**.

Interprocess communication

Most parallel computing tasks require communication between processes. This can be accomplished in several different ways on a unix system. The pipe, **|**, is a standard example:

```
% ps aux | grep a.out
```

The **ps** and **grep** processes are running in parallel and are communicating using a pipe. Data flows in one direction and the processes must have a common ancestor. The pipe handles synchronisation of data (**grep** must wait for data from **ps** and **ps** may not deliver data faster than **grep** can handle, for example).

The communication is usually local to one system, but using **rsh** (remote shell) or **ssh** (secure shell) it may be possible to communicate between different computers:

```
% ps aux | ssh other_computer "grep a.out > /tmp/junk"
```

/tmp/junk is created on **other_computer**. (There are other remote commands such as **rcp/scp**, remote copy).

FIFOs (or named pipes) can be used to communicate between two unrelated processes. A general way to communicate between computers over a network is to use so called sockets.

When a (parallel) computer has shared memory it is possible to communicate via the memory. Two (or more processes) can share a portion of the memory. Here comes a master (parent) program.

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

int main()
{
  int              exit_stat, shmid, info, k;
  pid_t            pid;
  struct shmid_ds buf;
  double          *shmaddr;
  char             s_shmid[10];
  /*
   * Create new shared memory segment and then
   * attach it to the address space of the process.
   */
  shmid=shmget(IPC_PRIVATE, (size_t) 512, SHM_R|SHM_W);
  shmaddr = shmat(shmid, (void*) 0, 0);

  /* Store some values */
  for (k = 0; k < 512 / 8; k++)
    *(shmaddr + k) = k;

  /* Create new proces */
  if ((pid = fork()) < 0) {
    printf("fork error\n");
    return 1;
  } else if (pid == 0) {        /* I am a child */
```



```
    /* convert int to string */
    sprintf(s_shmid, "%d", shmid);

    if (execlp("./child", "child_name", s_shmid,
               (char *) 0) < 0) {
      printf("*** In main: execlp error.\n");
      return 1;
    }
  } else {
    wait(&exit_stat);
    /* Remove the segment. */
    info = shmctl(shmid, IPC_RMID, &buf);
  }
  return 0;
}
```

Here comes a slave (child) program.

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

int main(int argc, char *argv[])
{ int              k, shmid;
  double          *shmaddr;

  printf("In child\n"); printf("argc = %d\n", argc);
  printf("argv[0] = %s\nargv[1] = %s\n",argv[0],argv[1]

  shmid = atoi(argv[1]);        /* convert to int */
  printf("shmid   = %d\n", shmid);
  shmaddr = shmat(shmid, (void*) 0, SHM_RDONLY);

  for (k = 0; k < 5; k++) /* "Fetch" and print values */
    printf("*(shmaddr+%d) = %f\n", k, *(shmaddr + k));
  return 0;
}
```



```
% gcc -o master master.c
% gcc -o child child.c
% master
In child
argc = 2
argv[0] = child_name
argv[1] = 22183946
shmid   = 22183946
*(shmaddr+0) = 0.000000
*(shmaddr+1) = 1.000000
*(shmaddr+2) = 2.000000
*(shmaddr+3) = 3.000000
*(shmaddr+4) = 4.000000
```

In general some kind of synchronisation must be used when accessing the memory. There are such tools (e.g. semaphores) but since we will look at a similar construction in the next section we drop the subject for now.

Using the command **ipcs** we can get a list of segments. It may look like:

```
% ipcs
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      r
status
0x00000000 22249482   thomas    600        512        0

... more stuff
```

In case of problems we can remove segments, e.g.

```
ipcrm -m 22249482
```

Nonblocking communication - a small example

Suppose we have a pool of tasks where the amount of time to complete a task is unpredictable and varies between tasks.
We want to write an MPI-program, where each process will ask the master-process for a task, complete it, and then go back and ask for more work. Let us also assume that the tasks can be finished in any order, and that the task can be defined by a single integer and the result is an integer as well (to simplify the coding).
The master will perform other work, interfacing with the user, doing some computation etc. while waiting for the tasks to be finished.
We could divide all the tasks between the processes at the beginning, but that may lead to load inbalance.
An alternative to the solution, on the next page, is to create two threads in the master process. One thread handles the communication with the slaves and the other thread takes care of the user interface.
One has to very careful when mixing threads and MPI, since the MPI-system may not be thread safe, or not completely thread safe. The MPI-2.0 standard defines the following four levels:

- **MPI_THREAD_SINGLE** Only one thread will execute.
- **MPI_THREAD_FUNNELED** The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are "funneled" to the main thread).
- **MPI_THREAD_SERIALIZED** The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are "serialized").
- **MPI_THREAD_MULTIPLE** Multiple threads may call MPI, with no restrictions.

See the standard for more details.

The master does the following:

```
set_of_tasks = { task_id:s }

Send a task_id to each slave and remove
these task_id:s from set_of_tasks

while ( not all results have been received ) {
  while ( no slave has reported a result )  // NB
    do some, but not too much, work

  if ( tasks remaining ) {
    pick a task_id from the set_of_tasks and
    remove it from the set_of_tasks
    send task_id to the slave
    (i.e. to the slave that reported the result)
  } else
    send task_id = QUIT to slave
}
```

Here is the slave code:

```
dont_stop = 1  /* continue is a keyword in C*/
while ( dont_stop ) {
  wait for task_id from master
  dont_stop = task_id != QUIT

  if ( dont_stop ) {
    work on the task
    send result to master
  }
}
```

The nonblocking communication is used in the **while**-loop marked **NB**. If the master is doing too much work in the loop, in may delay the slaves.

Details about nonblocking communication

A nonblocking send start call initiates the send operation, but does not complete it. The send start call will return before the message was copied out of the send buffer. A separate send complete call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer.

Similarly, a nonblocking receive start call initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer. A separate receive complete call is needed to complete the receive operation and verify that the data has been received into the receive buffer.

This is where the master can do some work in parallel with the wait. Using a blocking receive the master could not work in parallel.

If the send mode is standard then the send-complete call may return before a matching receive occurred, if the message is buffered. On the other hand, the send-complete may not complete until a matching receive occurred, and the message was copied into the receive buffer.

Nonblocking sends can be matched with blocking receives, and vice-versa.

Here is comes a nonblocking send:

```
MPI_Request request;

MPI_Isend(&message, msg_len, MPI_INT, rank, tag,
          MPI_COMM_WORLD, &request);
```

It looks very much like a blocking send, the only differences are the name **MPI_Isend** (**I** stands for an almost immediate return), and the extra parameter, **request**. The variable is a handle to a so-called opaque object.

Think of ths communication object as being a C-structure with variables keeping track of the tag and destination etc. **request** is used to identify communication operations and match the operation that initiates the communication with the operation that terminates it. We are not supposed to access the information in the object, and its contents is not standardised.

A nonblocking receive may look like:

```
MPI_Request request;

MPI_Irecv(&message, msg_len, MPI_INT, rank,
          tag, MPI_COMM_WORLD, &request);
```

Here are some functions for completing a call:

```
MPI_Request request, requests[count];
MPI_Status  status;

MPI_Wait(&request, &status);
MPI_Test(&request, &flag, &status);
MPI_Testany(count, requests, &index, &flag, &status);
```

and here is a simplified description. **request** is a handle to a communication object, referred to as object.

MPI_Wait returns when the operation identified by **request** is complete. So it is like a blocking wait. If the object was created by a nonblocking send or receive call, then the object is deallocated and **request** is set to **MPI_REQUEST_NULL**

MPI_Test returns **flag = true** if the operation identified by **request** is complete. In such a case, **status** contains information on the completed operation; if the object was created by a nonblocking send or receive, then it is deallocated and **request** is set to **MPI_REQUEST_NULL**
The call returns **flag = false**, otherwise. In this case, the value of **status** is undefined.

Finally **MPI_Testany**. If the array of requests contains active handles then the execution of **MPI_Testany** has the same effect as the execution of

```
MPI_Test( &requests[i], flag, status),
  for i=0, 1 ,..., count-1,
```

in some arbitrary order, until one call returns **flag = true**, or all fail. In the former case, **index** is set to the last value of **i**, and in the latter case, it is set to **MPI_UNDEFINED**.

If **request** (or **requests**) does not correspond to an ongoing operation, the routines return immediately.

Now it is time for the example. We have **n_slaves** numbered from 0 up to **n_procs - 2**. The master has rank **n_procs - 1**. The number of tasks are **n_tasks** and we assume that the number of slaves is not greater than the number of tasks. **task_ids** is an array containing a non-negative integer identifying the task. A task id of **QUIT = -1** tells the slave to finish.

The computed results (integers) are returned in the array **results**.

next_task points to the next task in **task_ids** and **n_received** keeps track of how many tasks have been finished by the slaves.

Here comes the code. First the master-routine.


```
void master_proc(int n_procs, int n_slaves, int n_tasks
                 int task_ids[], int results[])
{
  const int max_slaves = 10, tag = 1, msg_len = 1;
  int hit, message, n_received, slave, next_task, flag;
  double d;
  MPI_Request requests[max_slaves];
  MPI_Status status;

  next_task = n_received = 0;

  /* Initial distribution of tasks*/
  for (slave = 0; slave < n_slaves; slave++) {
    MPI_Send(&task_ids[next_task], msg_len, MPI_INT,
             slave, tag, MPI_COMM_WORLD);

    /* Start a nonblocking receive*/
    MPI_Irecv(&results[next_task], msg_len, MPI_INT,
              MPI_ANY_SOURCE, MPI_ANY_TAG,
              MPI_COMM_WORLD, &requests[slave]);
    next_task++;
  }

  /* Wait for all results to come in ...*/
  while (n_received < n_tasks) {
    flag = 0;
    while (!flag) {
      /* Complete the receive */
      MPI_Testany(n_slaves, requests, &hit, &flag,
                  &status);
      d = master_work();  /* Do some work */
    }
```



```
    n_received++;               /* Got one result */
    slave = status.MPI_SOURCE;  /* from where?    */

    /* Hand out a new task to the slave,
       unless we are done
     */
    if (next_task < n_tasks) {
      MPI_Send(&task_ids[next_task], msg_len, MPI_INT,
               slave, tag, MPI_COMM_WORLD);

      MPI_Irecv(&results[next_task], msg_len, MPI_INT,
                MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &requests[hit]);
      next_task++;
    } else {  /* No more tasks */
      message = QUIT;
      MPI_Send(&message, msg_len, MPI_INT, slave, tag,
               MPI_COMM_WORLD);
    }
  }
}
```


and then the code for the slaves

```
void slave_proc(int my_rank, int master)
{
  const int msg_len = 1, tag = 1;
  int message, result, dont_stop;
  MPI_Status status;

  dont_stop = 1;
  while (dont_stop) {
    MPI_Recv(&message, msg_len, MPI_INT, master,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    dont_stop = message != QUIT;
    if (dont_stop) {
      /* Simulate work */
      result = 100 * message + my_rank;
      sleep(message);

      MPI_Send(&result, msg_len, MPI_INT, master,
               tag, MPI_COMM_WORLD);
    }
  }
}
```

Suppose we are using three slaves and have ten tasks, the **`task_ids`**-array takes indices from zero to nine.

The work is simulated by using the **`sleep`**-function and the ten tasks correspond to sleeping 1, 2, 3, 1, 2, 3, 1, 2, 3, 1 seconds. The work done by the master, in **`master_work`**, takes 0.12 s per call.

The table below shows the results from one run.
When a number is repeated two times the slave worked with this task for two seconds (similarly for a repetition of three).

```
       slaves          task number     sleep time
      0   1   2            0               1
time                       1               2
1     0   1   2            2               3
2     3   1   2            3               1
4     5   4   2            4               2
4     5   4   6            5               3
5     5   7   8            6               1
6     9   7   8            7               2
7             8            8               3
                           9               1
```

So had it been optimal, the run should have taken 7 s wallclock time (the sum of the times is 19, so it must take more than 6 s wallclock time, as $3 \cdot 6 < 19$. The optimal time must be an integer, and the next is 7). The time needed was 7.5 s and the master was essentially working all this time as well.

Using two slaves the optimal time is 10 s, and the run took 10.8 s.