

Perl for Biologists

Session 13

May 27, 2015

Parallelization with Perl

Jaroslav Pillardy

Session 12 Exercises Review

1. Write a script that checks for new versions of a set of software titles listed in a file. Choose 5 programs from BioHPC Lab software list (<http://cbsu.tc.cornell.edu//lab/labsoftware.aspx>) and implement them. Check for new versions on the original software websites, NOT BioHPC website.

HINT: You can use script4.pl as the starting point

HINT: You will need to put URL and tag delimiters info for each program in the input file.

Software chosen:

BWA

Bioconductor

LUCY2

Picard

STRUCTURE

Session 12 Exercises Review

Configuration file (exercise1.config) is tab separated text file and hold necessary information for searching URLs for each program

name<TAB>URL<TAB>tag1<TAB>offset1<TAB>tag2<TAB>offset2<TAB>last

name	the name of the program
URL	URL of the page with version info
tag1	string to find the beginning of version region
offset1	how many characters to skip from the beginning of tag1
tag2	string ending the version region
offset2	how many characters to skip (left or right) from tag2
last	do we search for first (last=0) or last (last=1) occurrence

```
#!/usr/local/bin/perl
use LWP;

#what is our current version?
my %versions;
open in, "exercisel.data";
while(my $vtmp = <in>)
{
    chomp $vtmp;
    my ($programe, $pver) = split /\t/, $vtmp;
    $versions{$programe} = $pver;
}
close(in);
my $message = "";
my $subject = "";

my $ua = LWP::UserAgent->new;
$ua->agent("MyApp/0.1 ");
open in, "exercisel.config";
while(my $entry = <in>)
{
    if($entry =~ /^#/){next;}
    print "checking $name\n";
    chomp $entry;
    my ($name, $url, $tag1, $soff1, $tag2, $soff2, $last) = split /\t/, $entry;
    my $req = HTTP::Request->new(GET => $url);
    $req->header(Accept => "text/html, */*;q=0.1");
    my $res = $ua->request($req);
}
```

```

if ($res->is_success)
{
    my $n = index($res->content,$tag1);
    if($last == 1){$n=rindex($res->content, $tag1);}
    if($n == -1)
    {
        $message .= "ERROR: Cannot find version string for $name\n";
        print "ERROR: Cannot find version string for $name\n";
    }
    else
    {
        my $current = substr($res->content, $n+$off1);
        $current = substr($current, 0, index($current, $tag2) - $off2);
        if($current ne $versions{$name})
        {
            $message .= "New $name version found: $current\n";
            print "New $name version found: $current\n";
            $versions{$name} = $current;
        }
    }
}
else
{
    $message .= "ERROR: Cannot open $name web page $url\n";
    print "ERROR: Cannot open $name web page $url\n";
}
print "DONE\n";
}
close(in);

```

```
if($message ne "")
{
    open out, ">exercisel.data";
    foreach my $entry (keys %versions)
    {
        print out "$entry\t$versions{$entry}\n";
    }
    close(out);
    $subject = "Automated version checker mail";
    use Mail::Sendmail;
    %mail = (To => 'jarekpp@yahoo.com',
             From => 'jp86@cornell.edu',
             subject => $subject,
             Message => $message,
             smtp=>'appsmtplib.cornell.edu');
    sendmail(%mail);
}
```

Running programs: processes

Every running program is executed as a ***process***

Process is an object of UNIX (Linux) kernel (core of the system)

It is identified by process id (PID, an integer)

It has an allocated region in memory containing:

- code (binary instructions)
- execution queue(s) – set of pointers showing which instruction(s) to execute
- data (variables)
- copy of environment (variables, arguments etc.)
- communication handles (STDIN, STDOUT, file handles etc.)
- ...

Running programs: processes

OS kernel assigns processes to available cores dynamically, i.e. one process may share a core with another process by executing instructions in turns.

The more cores available the more processes run concurrently.

Unused processes can be swept out from memory to disk (swap space), and loaded when needed.

One can run many more processes than available cores, but if they all are CPU-intensive loss of performance will occur due to a cost of switching between processes.

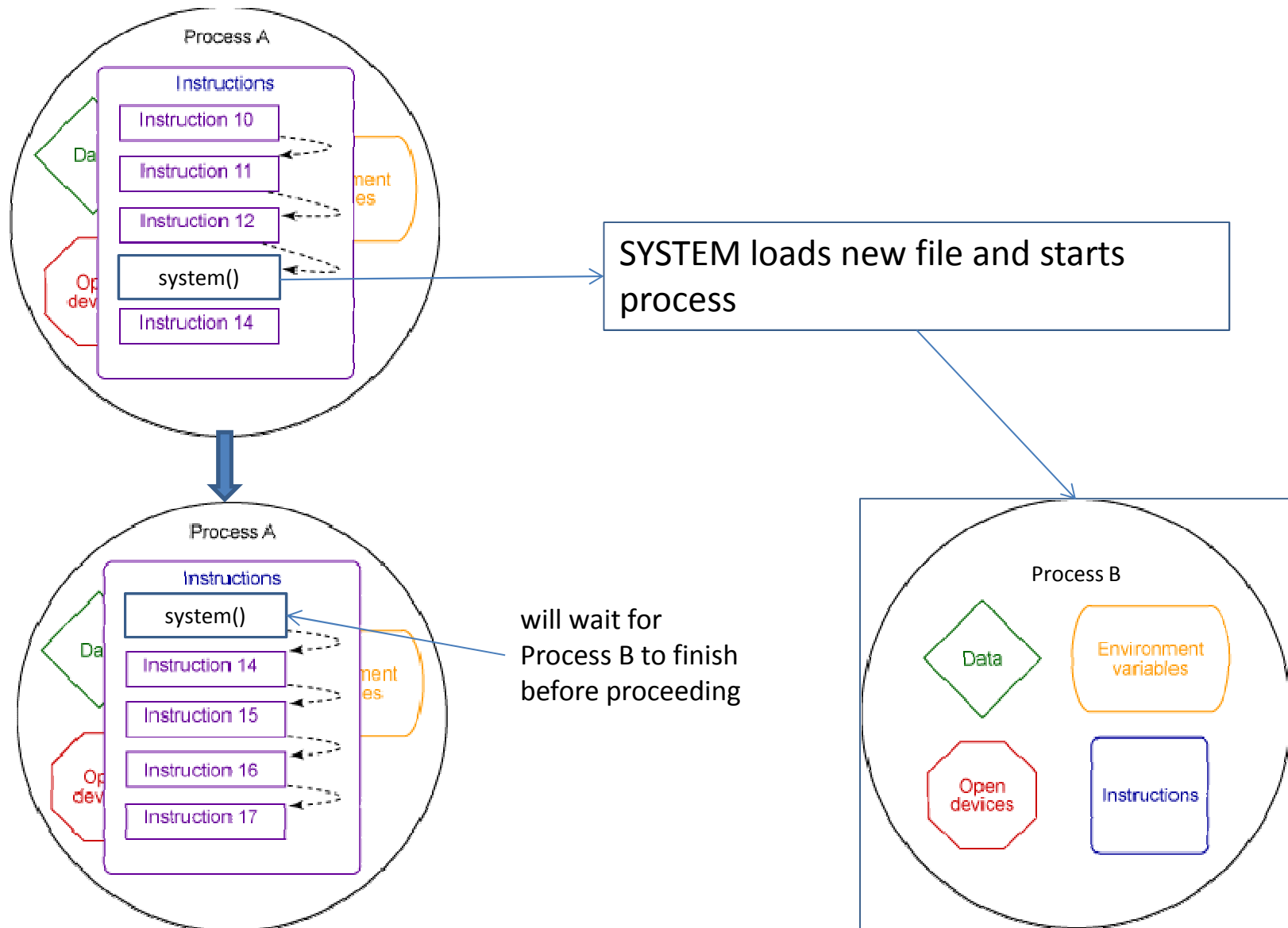
Running programs: processes

Process is always created by system, but the creation may be requested by another process (e.g. `system()` function).

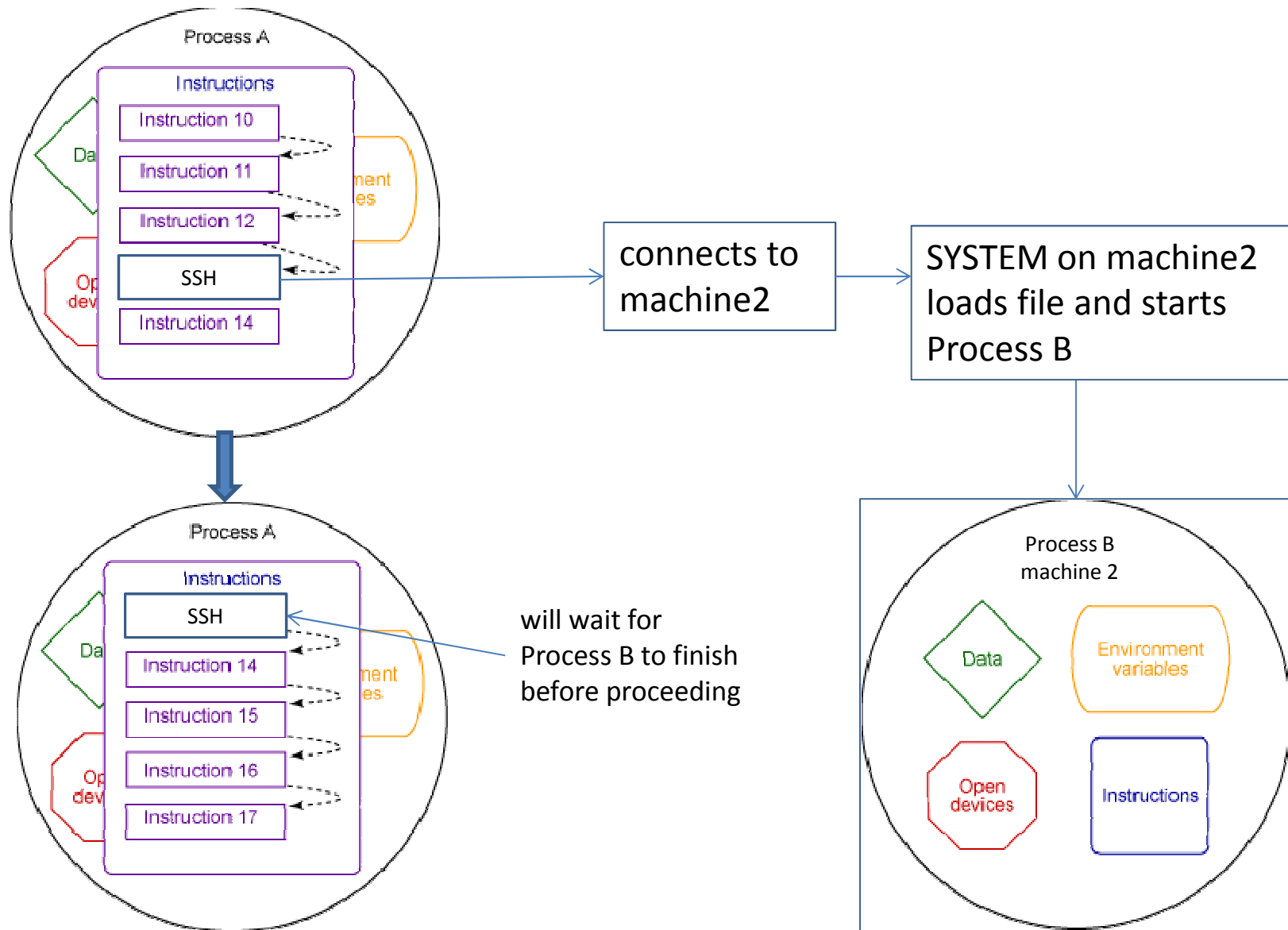
There are three ways for creating processes:

- Invoking a new program on the same machine
e.g. `system()`
- Invoking a new program on different (remote) machine
e.g. `ssh`
- Creating a clone of current process (always on the same machine)
`fork()` function

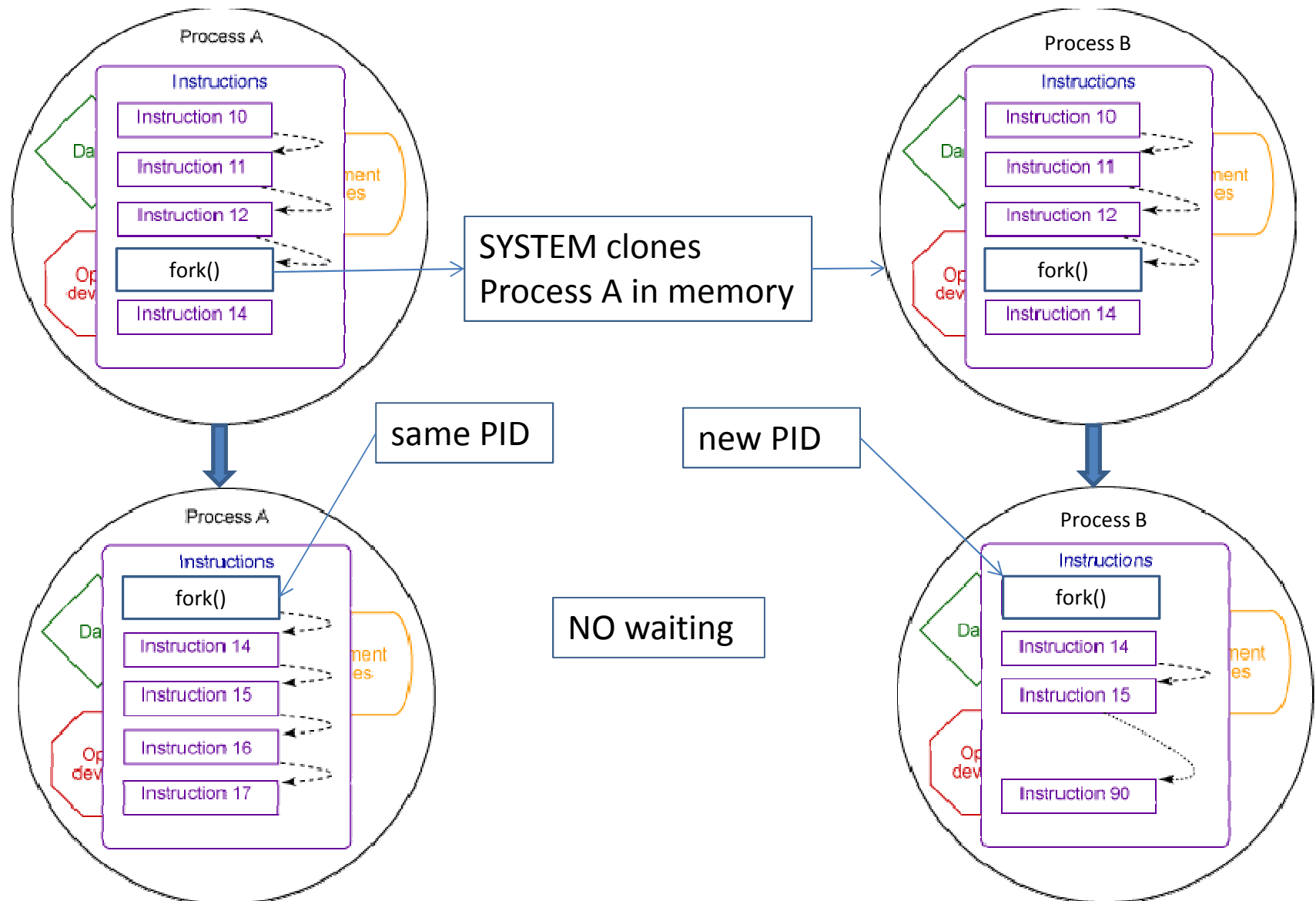
Starting processes with system()



Starting processes with ssh



Starting processes with fork()



Running programs: processes

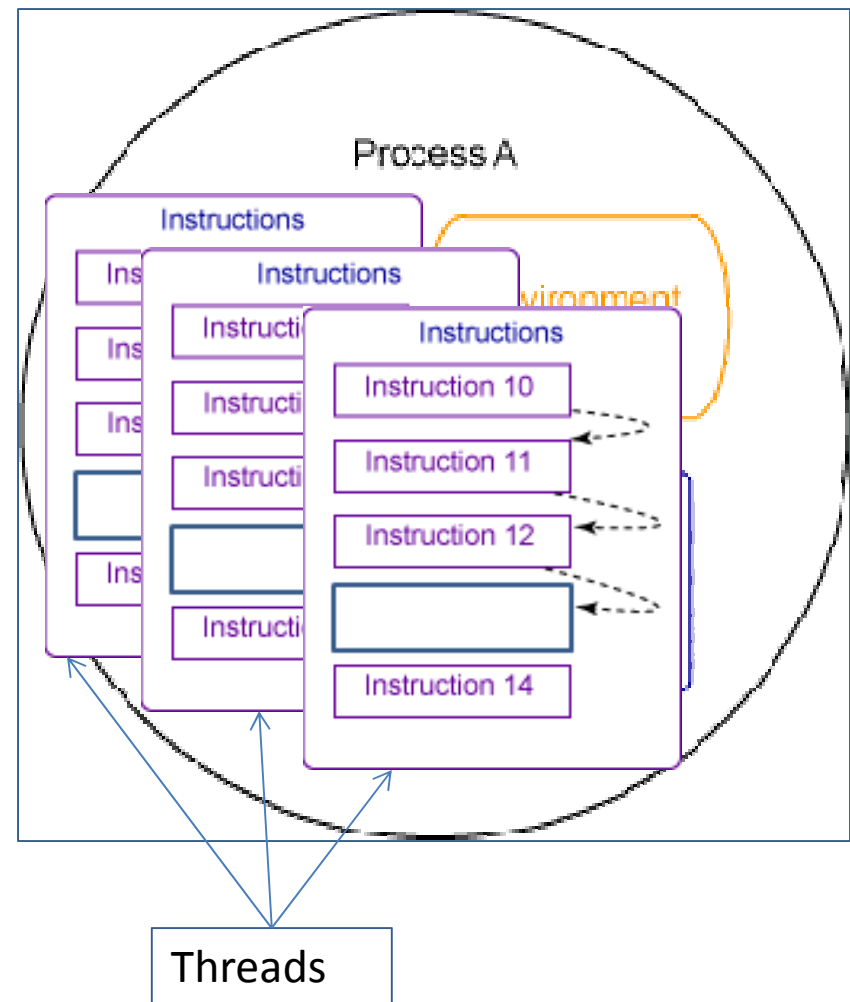
Processes may talk to each other using

- Pipes – output from one may be input to another
- Files – one program may write and other read the same file
- Signals - special, pre-defined, messages passed from one to another process via OS (SIGINT = CTRL-C, SIGTERM, SIGKILL etc.).
- Special libraries - there are libraries specializing in inter-process communication (Message Passing Interface, MPI)

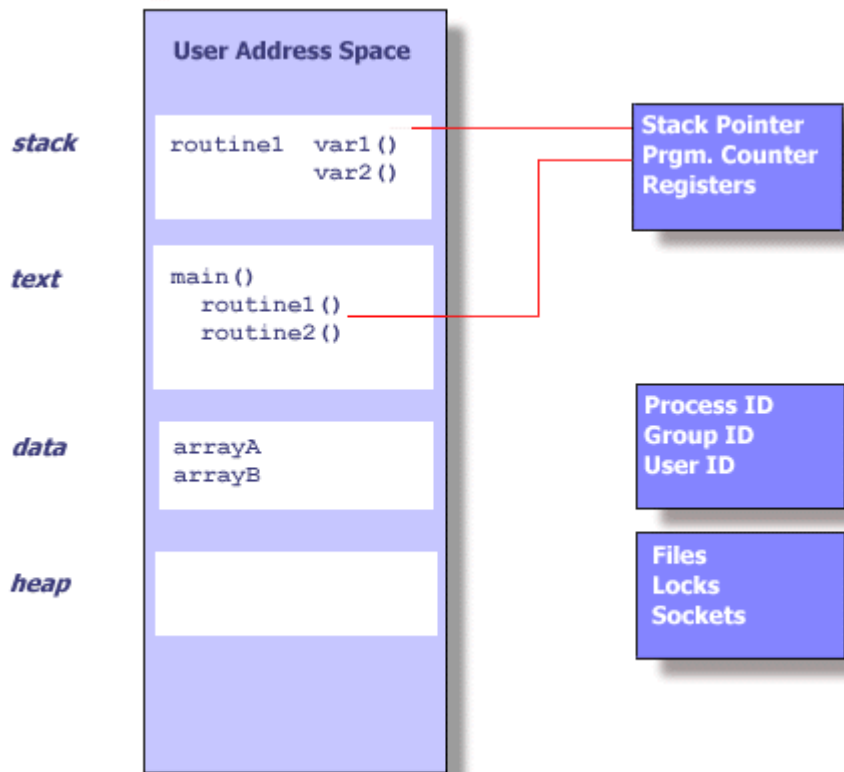
Threads

There is a way to execute more than one set of commands inside the same process – by using more than one execution queue.

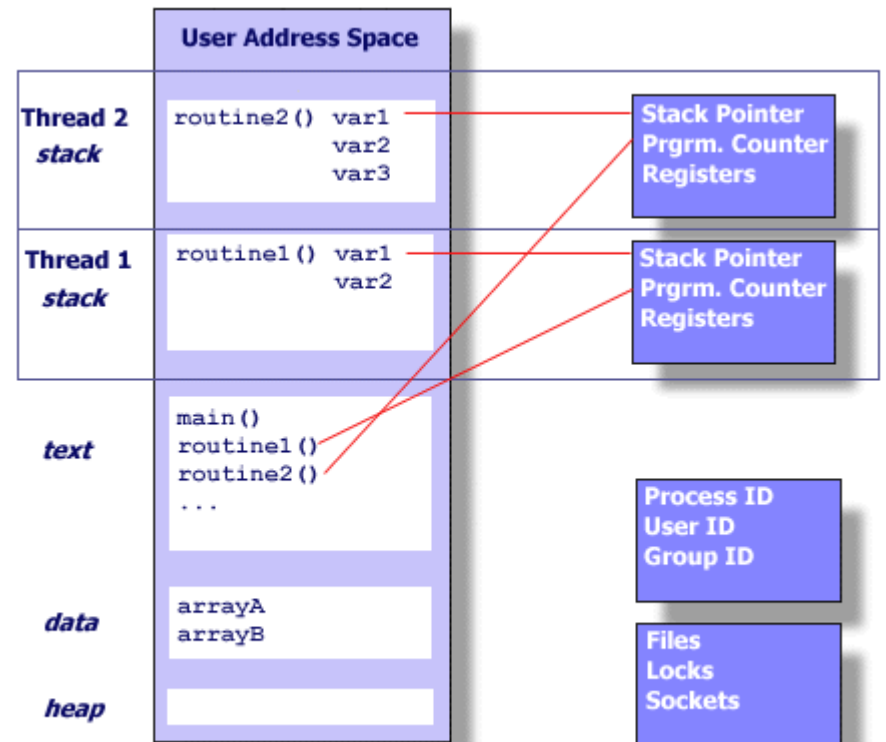
Each separate execution queue is a **thread**, they all have access to the same memory, environment etc. inside the process.



Threads



single thread process



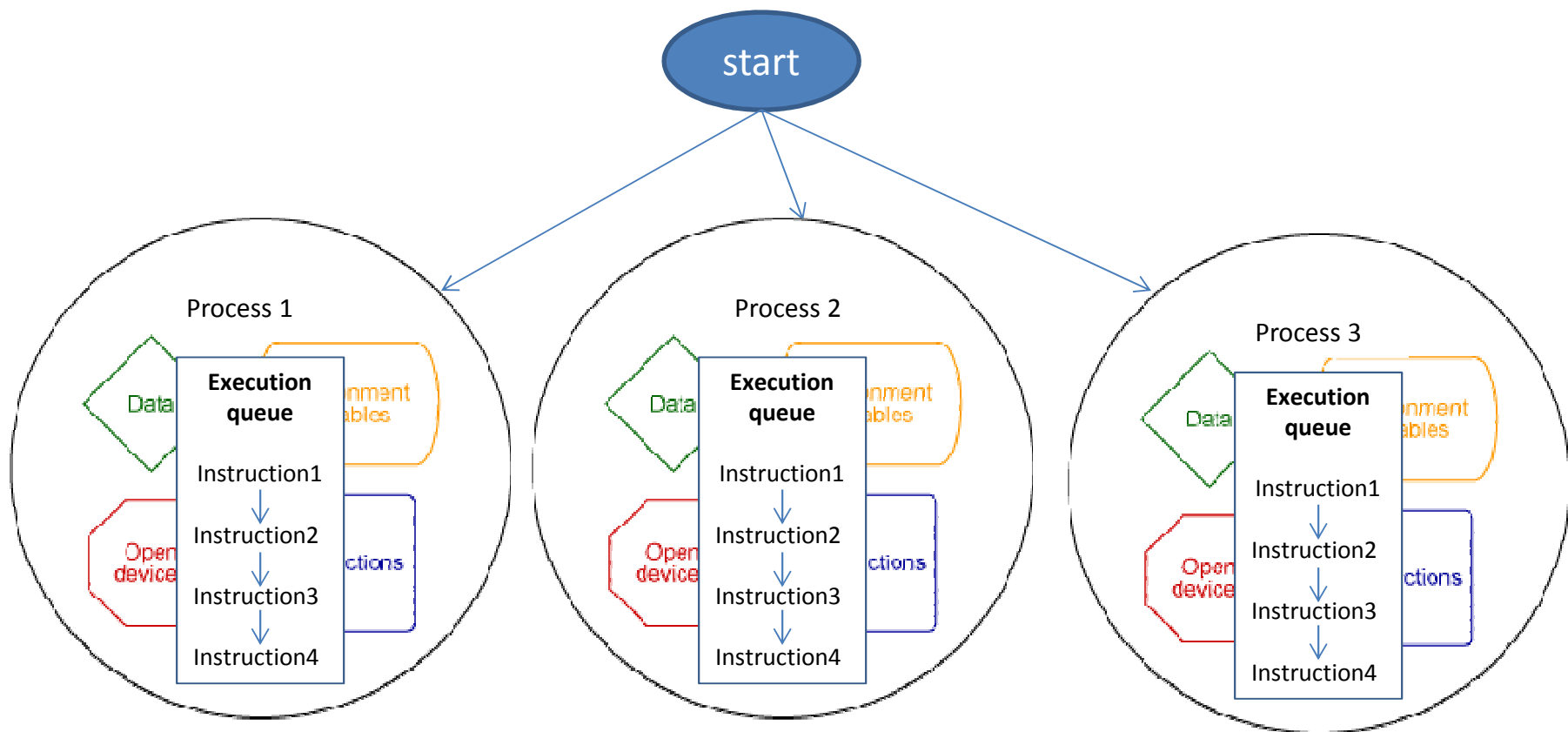
multithread process

Copied from <https://computing.llnl.gov/tutorials/pthreads/>

Parallel execution

Multi-processing (distributed):

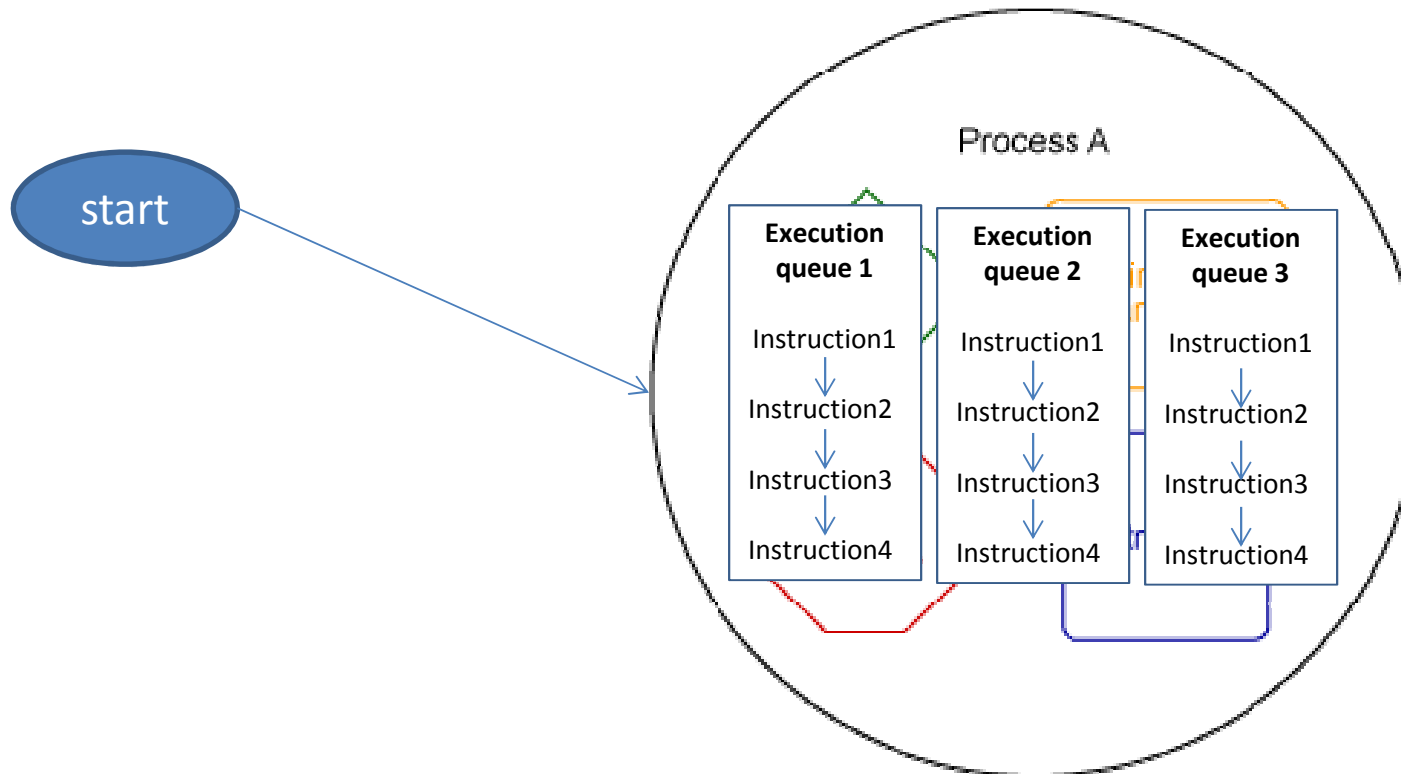
Multiple separate processes on the same or different machines coordinating their execution via passing data or messages. They cannot read each others memory.



Parallel execution

Multi-threading (shared memory):

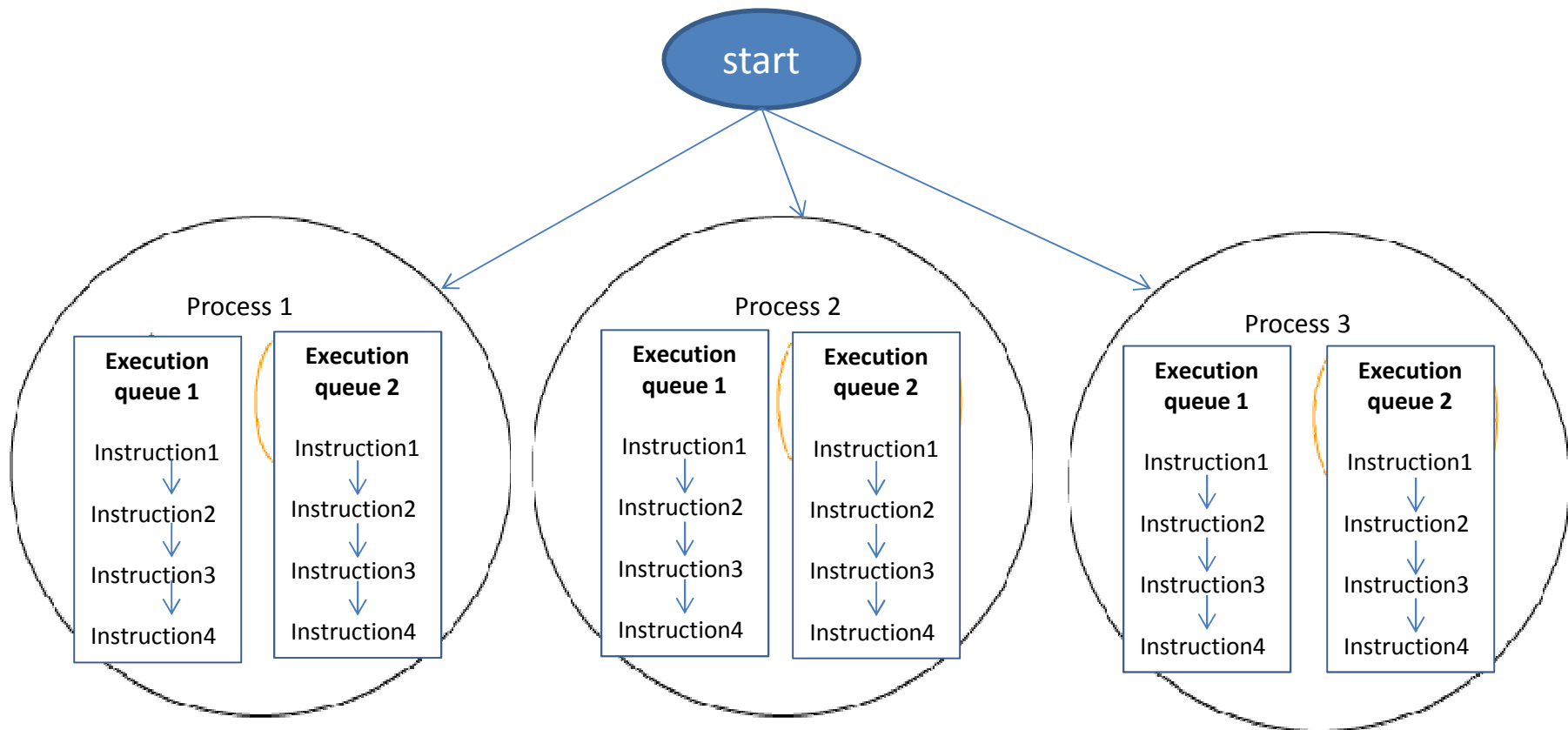
There is only one process with multiple execution queues inside. Threads communicate via messages or memory access – each thread has access to the same memory area.



Parallel execution

Multi-processing (distributed) with multi-threading :

Multiple separate processes on the same or different machines coordinating their execution via passing data or messages, each multithreaded.



Parallel execution

Deciding between multithreading and multiprocessing usually comes down to I/O versus CPU requirements.

CPU intensive

Very good for same machine multiprocessing.

I/O intensive

Good for same machine multithreading, or multiprocessing on SEPARATE machines (each of them does its own I/O).

I/O intensive and CPU intensive

Best for multiprocessing on SEPARATE machines, each process multithreaded.

fork()

This function clones the current process – the new process is identical to the previous one, except it has a new PID.

On the original process (master) the function returns the new PID of the new process (child).

On the new process (child) the function returns 0.

On error the function returns -1.

fork()

```
my $pid = fork();
if($pid < 0)
{
    #error
    print "ERROR: Cannot fork $!\n";
    #further error handling code
    exit;
}
elsif($pid == 0)
{
    #child code
    child_exec();
    exit;
}
else
{
    #master code
    master_exec();
    exit;
}
```

multi-process parallelization example: same machine

- we have a set of tasks that do not communicate with each other during execution
- our program should execute them on one machine in parallel, with maximum number of processes limited to a pre-defined number
- for this example will use a toy child execution: each child should sleep a random number of seconds between 5 and 20.

multi-process parallelization example: algorithm

- Master process will be devoted to process control
- Master will create child processes, up to maximum allowed limit
- Child processes will execute the “work” part
- Master will monitor child processes, when a child process finishes, master will create another child process if there are unprocessed tasks left

waitpid()

This function is used to monitor condition of child process

```
my $result = waitpid($pid, WNOHANG) ;
```

WNOHANG flag tells **waitpid** to return the status of the child process with given pid `$pid` WITHOUT waiting for this process to complete.

If the return value is less or equal 0 the child process still exists and is running.

Otherwise the return value is the completion code of the process – same as `system()`.


```
#!/usr/local/bin/perl
```

```
use POSIX ":sys_wait_h";
```

need this module
for fork()

```
my $maxtask = 8;
```

```
my $ntasks = 15;
```

```
#initial fork child processes
```

```
my @procs;
```

```
my $task = 0;
```

```
print "STARTING: maxtask=$maxtask ntasks=$ntasks\n";
```

```
for(my $i=0; $i<$maxtask; $i++)
```

```
{
```

```
    $task++;
```

```
    print "starting child $i task $task ";
```

```
    $procs[$i] = start_task($i, $task, @procs);
```

```
    print " pid $procs[$i]\n";
```

```
}
```

```
#waiting for child processes to finish and execute remaining tasks in their place
```

```

while(1)
{
    sleep(1); #there is no need to check every millisecond - it would use too much CPU
    my $n=0;
    for(my $i=0; $i<=$#procs; $i++)
    {
        if($procs[$i] != 0)
        {
            my $kid = waitpid($procs[$i], WNOHANG);
            if($kid <= 0)
            {
                $n++; #process exists
            }
            else
            {
                print "Child " . ($i+1) . " finished (pid=" . $procs[$i] . ")\n";
                $procs[$i] = 0 ;
                if($task < $ntasks)
                {
                    $task++;
                    $procs[$i] = start_task($i, $task, @procs);
                    print " child " . ($i+1) . " restarted for task $task with pid $procs[$i] \n";
                    $n++;
                }
            }
        }
    }
    if($n==0) {last;}
}
print "ALL DONE\n";

```

waitpid() checks the status of a child process, WNOHANG flag means no waiting – just immediate status return

```
sub start_task
{
    my ($i, $task, @procs) = @_ ;

    my $pid = fork() ;
    if($pid < 0)
    {
        #error
        print "\n\nERROR: Cannot fork child $i\n";
        for(my $j=0; $j<=$#procs; $j++)
        {
            system("kill -9 " . $procs[$j]);
        }
        exit;
    }
    if($pid == 0)
    {
        #child code
        child_exec($i+1, $task);
        exit;
    }
    #master - continue, $pid contains child pid
    return $pid;
}
```

```
sub child_exec
{
    my ($num, $tasknum) = @_;

    my $seed = time * $num * $num;
    srand($seed);

    $nsec = int(rand(20)) + 5;
    #print "CHILD $num: task number $tasknum, wait time $nsec sec\n";
    sleep($nsec);
}
```

multi-process parallelization example: same machine PAML simulation

- we have a set of genes and we want to run PAML program on each of them
- our program should execute them on one machine in parallel, with maximum number of processes limited to a pre-defined number
- The program needs to prepare a directory to run each gene, then execute PAML in parallel

multi-process parallelization example: same machine PAML simulation

The program should be run from local directory on the workstation, i.e. /workdir

Here is the script used to set up the local data

```
#!/bin/bash
mkdir /workdir/jarekp
cd /workdir/jarekp
tar -xzf /home/jarekp/perl_13/paml4.7.tgz
cp -R /home/jarekp/perl_13/seqdata .
cp /home/jarekp/perl_13/mytree.dnd .
cp /home/jarekp/perl_13/template.control .
```

```
#!/usr/local/bin/perl
use POSIX ":sys_wait_h";

my $maxtask = 8;
my $ntasks = 0;
my @tasklist;

#PAML parameters
my $pamlcmd = "/workdir/jarekp/paml4.7/bin/codeml";
my $template_control_file = "template.control";
my $tree = "mytree.dnd";
my $datadir = "seqdata";
my $outdir = "results";

if(! -e $template_control_file)
{
    print "Template control file is NOT available\n";
    exit;
}
if(! -e $tree)
{
    print "Tree file is NOT available\n";
    exit;
}
if(! -d $datadir)
{
    print "The data directory is NOT available\n";
    exit;
}
```

```
if(! -d $outdir)
{
    mkdir $outdir;
}

print "Preparing PAML and input files\n";

#put template control files in a variable.
#a new control file will be created for each gene
open (IN, $template_control_file)
    or die "The template control file $template_control_file cannot be opened";
my $template_control_content = "";
while (<IN>)
{
    $template_control_content .= $_;
}
close IN;

#get the list of files in seqdata
opendir(my $dh, $datadir) || die "can't opendir $datadir: $!";
my @alnfiles;
foreach my $entry (readdir($dh))
{
    if($entry =~ /\.phy$/)
    {
        push(@alnfiles, $entry);
    }
}
closedir $dh;
```



```
#prepare directories - one for each gene
#store their names in @tasklist
foreach my $alnfile (@alnfiles)
{
    my $seqid = $alnfile;
    $seqid =~ s/\..+//;

    my $outseqdir = $outdir. "/" . $seqid;
    if (! -d $outseqdir)
    {
        mkdir $outseqdir;
        system ("cp $tree $outseqdir/" );
        system ("cp $datadir/$alnfile $outseqdir/" );
    }
    my $mycontrolfile = $template_control_content;
    $mycontrolfile =~ s/XXXXXX/$alnfile/;
    open CC, ">$outseqdir/my.control";
    print CC $mycontrolfile;
    close CC;
    push(@tasklist, $outseqdir);
}

my $ntasks = $#tasklist + 1;
print "We have $ntasks genes to run\n";
```

```

#initial fork child processes
my @procs;
my $task = 0;
print "STARTING: maxtask=$maxtask ntasks=$ntasks\n";
for(my $i=0; $i<$maxtask; $i++)
{
    $task++;
    print "starting child " . ($i+1) . " task $task\n";
    $procs[$i] = start_task($tasklist[$task-1], $pamlcmd, @procs);
    print "        child " . ($i+1) . " task $task started with pid $procs[$i]\n";
}

#waiting for child processes to finish and execute remaining tasks in their place
while(1)
{
    sleep(1); #there is no need to check every millisecond-it would use too much CPU
    my $n=0;
    for(my $i=0; $i<=$#procs; $i++)
    {
        if($procs[$i] != 0)
        {
            my $kid = waitpid($procs[$i], WNOHANG);
            if($kid <= 0)
            {
                #process exists
                $n++;
            }
        }
    }
}

```

```

else
{
    print "Child " . ($i+1) . " finished (pid=" . $procs[$i] . ")\n";
    $procs[$i] = 0 ;
    if($task < $ntasks)
    {
        $task++;
        $procs[$i] = start_task($tasklist[$task-1], $pamlcmd, @procs);
        print " child " . ($i+1) . " restarted for task $task with pid $procs[$i]\n";
        $n++;
    }
}
}
}
if($n==0) {last;}
}

print "ALL DONE\n";

sub child_exec
{
    my ($outseqdir, $pamlcmd) = @_;

    chdir($outseqdir);
    system("$pamlcmd my.control >& log");
}

```

```
sub start_task
{
    my ($taskstr, $pamlcmd, @procs) = @_;

    my $pid = fork();
    if($pid < 0)
    {
        #error
        print "\n\nERROR: Cannot fork child $i\n";
        for(my $j=0; $j<=$#procs; $j++)
        {
            system("kill -9 " . $procs[$j]);
        }
        exit;
    }
    if($pid == 0)
    {
        #child code
        child_exec($taskstr, $pamlcmd);
        exit;
    }
    #master - continue, $pid contains child pid
    return $pid;
}
```

multi-process parallelization example: different machines BLAST search

- We have a set of sequences in one fasta file (sequences.fasta)
- Our program should BLAST them against RefSeq mammalian RNA on a pre-defined set of machines in parallel, each BLAST should use multiple cores on each machine (BLAST supports multithreading)
- The fasta file should be split into blocks, each containing 5 sequences
- Each block should be searched as one unit on each machine

ssh without password in BioHPC Lab (between workstations)

```
>ssh-keygen -t rsa
```

(use empty password when prompted)

```
>cat .ssh/id_rsa.pub >> .ssh/authorized_keys
```

```
>chmod 640 .ssh/authorized_keys
```

```
>chmod 700 .ssh
```

Before running parallel script make sure you can ssh from your master machine to all machines listed in the script

```
#!/usr/local/bin/perl
use POSIX ":sys_wait_h";

my $maxtask = 4;
my @machines = qw(localhost cbsum1c2b015 cbsum1c2b014 cbsum1c2b012);
my $ntasks = 0;
my @tasklist;

my $exe = "/home/jarekp/perl_13/script3exe.pl";
my $datadir = "/home/jarekp/perl_13/data3";
my $outdir = "/home/jarekp/perl_13/out3";
my $fastafile = "/home/jarekp/perl_13/sequences.fa";
my $blocksize = 5;
```

```
print "Splitting large fasta file\n";

open (IN, $fastafile) or die "Fasta file $fastafile cannot be opened";
my $scount = 0;
my $block = 0;
my $line = 0;
while (my $txt=<IN>)
{
    $line++;
    if($txt =~ /^>/){$scount++;}
    if(($scount-1)%$blocksize == 0 && ($line-1)%2 == 0)
    {
        if($block>0){close (BLOCK);}
        $block++;
        open BLOCK, ">$datadir/block$block";
        push(@tasklist, "block$block");
    }
    print BLOCK $txt;
}
close IN;
close BLOCK;

my $ntasks = $#tasklist + 1;
print "We have $ntasks blocks to blast\n";
```



```

#initial fork child processes
my @procs;
my $task = 0;
print "STARTING: maxtask=$maxtask ntasks=$ntasks\n";
for(my $i=0; $i<$maxtask; $i++)
{
    $task++;
    print "starting child " . ($i+1) . " task $task\n";
    $procs[$i] = start_task($tasklist[$task-1], $machines[$i], $exe, @procs);
    print "        child " . ($i+1) . " task $task started with pid $procs[$i]\n";
}

#waiting for child processes to finish and execute remaining tasks in their place
while(1)
{
    sleep(1); #there is no need to check every millisecond-it would use too much CPU
    my $n=0;
    for(my $i=0; $i<=$#procs; $i++)
    {
        if($procs[$i] != 0)
        {
            my $kid = waitpid($procs[$i], WNOHANG);
            if($kid <= 0)
            {
                #process exists
                $n++;
            }
        }
    }
}

```

```
else
{
    print "Child " . ($i+1) . " finished (pid=" . $procs[$i] . ")\n";
    $procs[$i] = 0 ;
    if($task < $ntasks)
    {
        $task++;
        $procs[$i] = start_task($tasklist[$task-1], $machines[$i], $exe, @procs);
        print " child " . ($i+1) . " restarted for task $task with pid $procs[$i]\n";
        $n++;
    }
}
}
}
if($n==0) {last;}
}

print "ALL DONE\n";
```

```
sub start_task
{
    my ($datafile, $machine, $cmd, @procs) = @_;

    my $pid = fork();
    if($pid < 0)
    {
        #error
        print "\n\nERROR: Cannot fork child $i\n";
        for(my $j=0; $j<=$#procs; $j++)
        {
            system("kill -9 " . $procs[$j]);
        }
        exit;
    }
    if($pid == 0)
    {
        #child code
        child_exec($datafile, $machine, $cmd);
        exit;
    }
    #master - continue, $pid contains child pid
    return $pid;
}
```

```
sub child_exec
{
    my ($datafile, $machine, $cmd) = @_;

    if($machine eq "localhost")
    {
        system("$cmd $datafile");
    }
    else
    {
        system("ssh $machine $cmd $datafile");
    }
}
```

```
#!/usr/local/bin/perl

my $seqfile = $ARGV[0];

my $tempdir = "/workdir/jarekp";
my $local_dbdir = "/workdir/jarekp/db";
my $dbname = "RefSeq_mammalian.rna";
my $orig_dbdir = "/shared_data/genome_db/BLAST_NCBI";
my $datadir = "/home/jarekp/perl_13/data3";
my $outdir = "/home/jarekp/perl_13/out3";

if(! -e $tempdir){mkdir $tempdir;}
if(! -e $local_dbdir){mkdir $local_dbdir;}
if(! -e "$local_dbdir/$dbname.nal")
    {system("cp -f $orig_dbdir/$dbname.* $local_dbdir");}

my $temprundir = $tempdir . "/" . $seqfile;
mkdir $temprundir;
chdir($temprundir);
system("cp $datadir/$seqfile .");
system("blastall -a 8 -d $local_dbdir/$dbname -i $seqfile -o $seqfile.out -p
blastn >\& $seqfile.log");
system("cp $seqfile.out $outdir");
system("cp $seqfile.log $outdir");
chdir("../");
system("rm -rf $seqfile");
```

Exercises

1. Modify script1.pl so it executes commands (tasks) from a file (one task per line), each in separate directory. Print out total execution time of all tasks and average execution time per task (in seconds).