Processes

Alex Delis
NYU-Abu Dhabi

February 2021

The Concept of Process

■ A Process is an executable program while it is being run by the *CPU*.

$\Rightarrow$ Processes are also known as *jobs*, *tasks* and:

- have features that can be used.
- can be manipulated by the kernel to assist in the execution of executables.
- can communicate with others providing *interprocess communications (IPC)*.
- are provided with a number of *IPC* options.

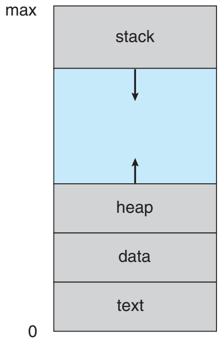
Process Concept

- As a program in execution, a process progresses sequentially towards its end.

- A process consists of multiple sections:
  - Text section which contains the program code.
  - Program counter (PC) and content of CPU registers.
  - Stack storing transient information about the run-time situation.
  - Data section that carries all global variables/structures.
  - Heap that entails dynamically allocated memory space while the program is in execution.

The Process Concept

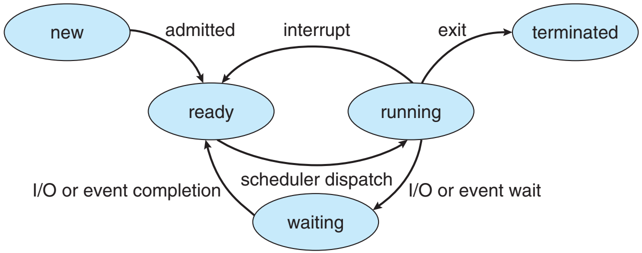
- A program until is loaded in main-memory is passive.
  - Once the program is loaded in MM becomes active.
  - The image of the loaded program in MM:



  - Execution starts either with $<CR>$ or with a mouse click on an GUI icon.
  - Once the program has the CPU is running.

States of a Process

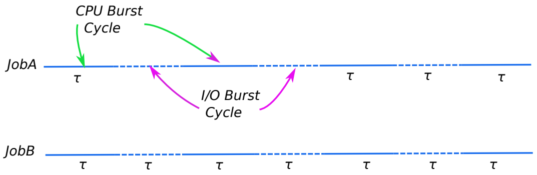
- A process may find herself in a number of states
  - new: process is being created
  - running: instructions are being executed by CPU
  - waiting: process awaits for an event or signal to occur
  - ready: process is waiting to be assigned to the CPU
  - terminated: process has completed execution.



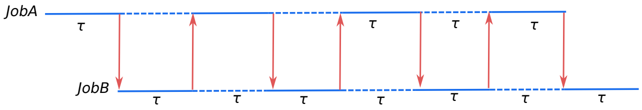
- Multiprogramming is about having multiple active programs in MM at the same time.
  - Multiprogramming enables the “CPU to be taken away” from one job and to be given to another task.
  - *CPU Utilization u* can be improved:
    - $u \rightarrow 1$
  - System Throughput can be also assisted/improved:
    - T = *amount of work / time needed to complete work*
    - Upper bound?

Batch and More Expedient CPU Processing

- Batch Processing for 2 jobs A and B.



- What if we could schedule things better?? Multiprogramming



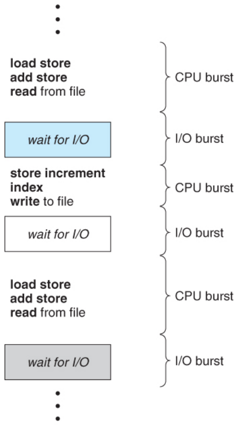
- Better time for job completion, throughput, utilization!

Benefits of Better Scheduling the CPU

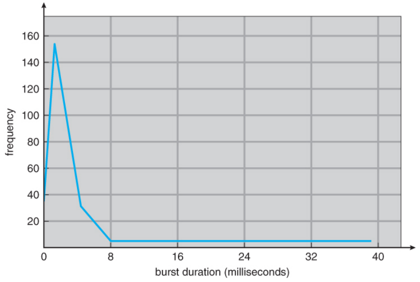
- By being *able* to multiplex CPU between 2 task the apparent benefits are:
  - *JobA* finishes earlier *but JobB* finishes in half time!
  - Throughput is improved by a factor of 2
  - Turnaround Time also improves to an average of 7.5τ
  - Time-multiplexing jobs with perfect requirements is unrealistic but points out to the benefits of multiprogramming.
- Question: under which circumstances and how multiprogramming can work??

CPU/IO Burst Cycles

- A CPU Burst Cycle is followed by an IO Burst Cycle until the end of the program.



- Frequency of the CPU Burst Length.



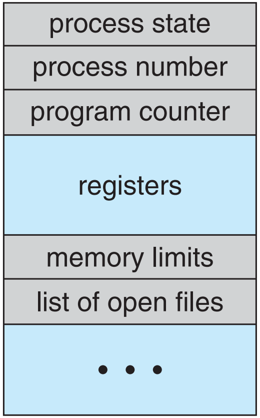
Characteristics of Processes

- Processes comes in mostly two types: CPU- and I/O-bound.
  - CPU-bound processes or hogs are tasks that they need mostly the CPU to complete their work; occasionally they may issue I/O operations.
    - Feature short I/O-bursts.
  - I/O-bound processes are jobs that expend most of their time in accomplishing I/O operations and much less in CPU-processing.
    - Feature short CPU-bursts.
- How to enable the switch of CPU between jobs resident in MM?

Process Table (or Process Control Block)

■ Every process has to have some identifying features that differentiate it from others that coexist in the system.

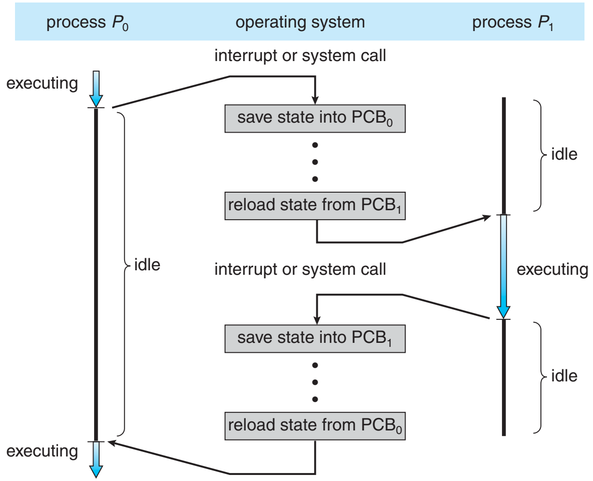
- Process ID: unique # id-ing a process
- Process state: running, waiting, etc
- PC: location of instruction to execute next
- CPU registers: content of all registers (when?)
- CPU scheduling information: priorities, scheduling queue pointers
- MM mgmt info: Mem allocated to process
- Accounting info: CPU used, clock time elapsed since start, time limits
- I/O status info: I/O devices allocated to process, list of open files



Management of Multiple PCBs

- PCBs are stored in kernel space.
  - How does the kernel manage the various PCBs?
    - They could be managed statically.
    - Dynamic management is more appropriate.
    - Linux uses doubly-linked list.
    - Hash-map could be used based on ProcessID

Switching Between Processes



Context Switching

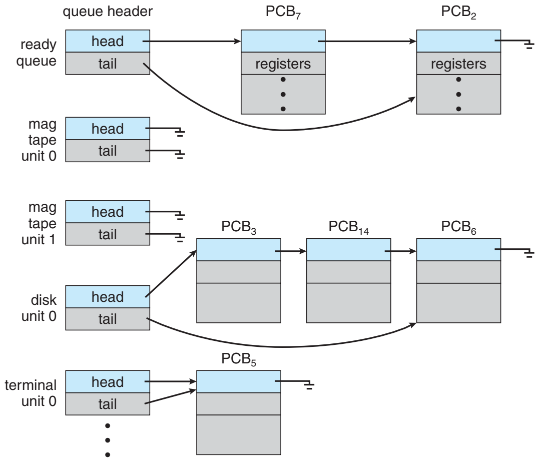
■ When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.

- Context of a process represented in PCB (i.e., snapshot).
- Context-switch time is *pure overhead* as kernel does no useful work.
- Context Switch time highly depends on hardware but in practice is not negligible.
- Some boards provides multiple sets of registers per CPU $\Rightarrow$ multiple contexts loaded at once.

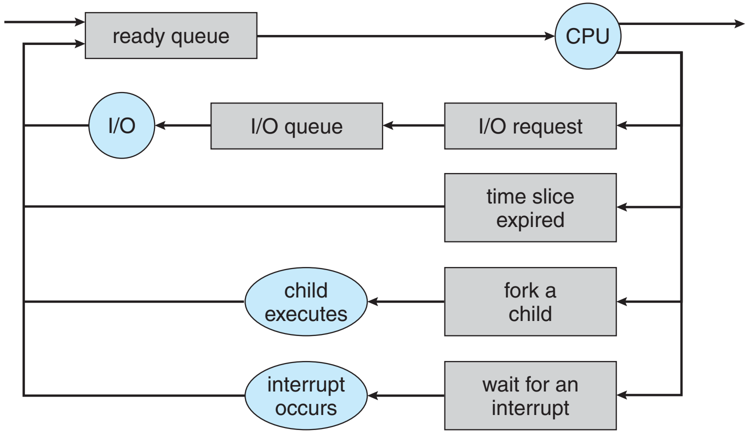
Process Scheduling

- Objective: maximize the CPU use i.e., *utilization*.
  - Process scheduler selects among the available (“ready”) processes one to surrender the CPU.
  - The kernel maintains a number of queues for processes:
    - Ready Queue: all processes that are memory resident and are ready to get the CPU.
    - Job Queue: set of all processes in system.
    - Device Queues: for each device there might be a set of processes awaiting for an I/O.
  - *Overall*: processes move from one queue to another until they complete execution.

State of Affairs in Queues



Queuing Diagram as a Job may Traverse its States



System Schedulers

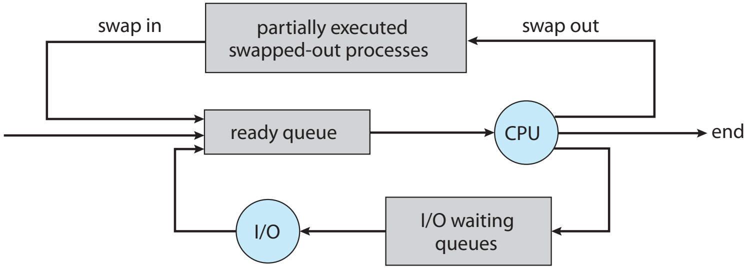
■ Scheduling decisions have to be taken not only wrt who is given the CPU but also on who can be admitted into the system.

- Short-term scheduler (STS) or CPU scheduler: selects which process should be executed next and allocates CPU with the help of *dispatcher*.
  - Short-term scheduler is invoked frequently i.e., every few milliseconds.
  - Must be fast.
  - It is sometimes the only scheduler in a system.
- Long-term scheduler (LTS) or job scheduler: selects which processes should be brought into the Ready Queue
  - Long-term scheduler is invoked infrequently i.e., every few seconds.
  - LTS controls the degree of multiprogramming.
  - Can be slow.

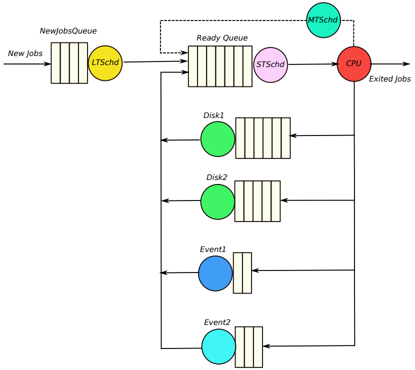
Medium-Term Scheduling

■ Medium-Term Scheduling (MTS) can be *added* if *degree of multiprogramming* has to be regulated.

- For example, due to long time spent with the CPU, a job has to be *swapped out* temporarily to offer others more ample opportunities to complete.



Where are all Schedulers being Placed?



- Dispatcher: Module that *gives control of CPU* to select process by Scheduler.
  - Loads the Registers and new PC.
  - Switches to user-mode by jumping to proper location of the user program (PC).
  - Must be really *fast*.

Background/Foreground Processes

- Most contemporary OSes can run tasks in both foreground/background mode.
  - Early mobile-OSes run 1 job in the foreground at the time due to display limited space.
  - This has been addresses as one can work with a program of choice in the foreground while multiple other services run “quietly” in the background at the same time.
  - In mobile-OSes, background jobs have often limited MM space and no interface.

Operations for Processes

- The kernel must provide fundamental mechanisms to:
  - create a new process,
  - terminate/abort a process,
  - have a process run a newly loadable address space (i.e., a new program),
  - ask a process to wait for an event to occur,
  - maintain basic communication through signaling with other processes.
  - etc. etc. etc.

Process Creation

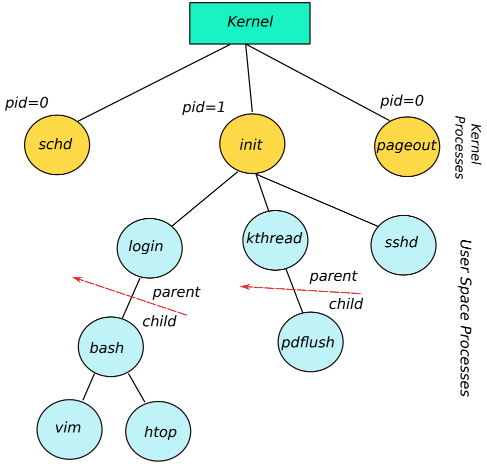
- The general spirit: a parent process creates 1 or more children which in turn may create other(s) essentially forming a tree of processes.
  - A process is always identified by a ProcessID that is unique.
  - How resources are being shared between a parent and a child (or children)?
    - Parent and children share all resources
    - Children share subset of parent's resources
    - Parent and child share no resources

Process Creation

- How does execution carries on when a child appears?
  - Parent and child may execute concurrently.
  - Parent waits until children terminate.
- How is the *address space* is managed among processes?
  - Child is a duplicate of parent
  - Child has a (new) program loaded into it.
- What happens in Linux?
  - `fork()` system call is used to create a new process.
  - The `execve()` system call is used to replace the address space of a newly-forked process with the memory space of a new program.

The Logical Hierarchy of Processes

- Processes are all anchored in a kernel process called init (the mother of all).



The `fork()` System Call

- The `fork()` system call does the following:
  - Generates a *copy* of the (parent) process which now becomes the child process.
  - The call returns *different* values:

    $\Rightarrow$ 0 to child process, and

    $\Rightarrow$ ***childPID*** to the parent process.
  - The kernel passes *most* of the parents's information to the child process.
  - After the call of `fork()`, both child and parent are concurrently running and continue their execution at the NEXT statment AFTER `fork()`

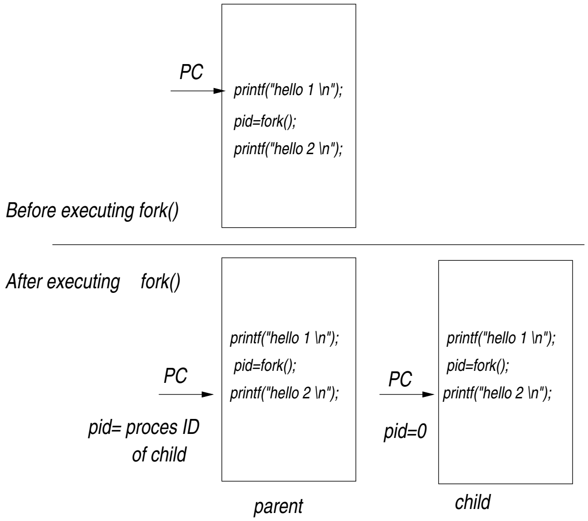
The `fork()` Call

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

int     main(void){
        pid_t childpid;

        childpid=fork();
        if (childpid== -1){
                perror("Failed␣to␣fork\n");
                exit(1); }

        if (childpid==0)
                printf("I␣am␣the␣child␣process␣␣with␣ID:␣%lu␣\n",(long)getpid());
        else    printf("I␣am␣the␣parent␣process␣with␣D:␣%lu␣\n",(long)getpid());
        return(0);
}
```

- The output is:

```
ad@rhodes:~/ForkExecSignals-v3$ ./1a-fork
I am the parent process with D: 46344
I am the child process  with ID: 46345
ad@rhodes:~/ForkExecSignals-v3$ ./1a-fork
I am the parent process with D: 46346
I am the child process  with ID: 46347
ad@rhodes:~/ForkExecSignals-v3$
```

What happens to PC after a `fork()`



Example with a `fork()`

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

#define         BUFFSIZE        20

int     main(void){
        int             i=0;
        static  char    bufferA[BUFFSIZE];
        static  char    bufferB[BUFFSIZE];

        if  (fork()==0){
                strcpy(bufferA,"In␣Child␣Process␣\n");
                write(1,bufferA,sizeof(bufferA));
                }
        else    {
                strcpy(bufferB,"In␣Parent␣Process␣\n");
                write(1,bufferB,sizeof(bufferB));
                }
}
```

```
ad@rhodes:~/ForkExecSignals-v3$ ./3-ParentChild2
In Parent Process
In Child Process
ad@rhodes:~/ForkExecSignals-v3$ ./3-ParentChild2
In Parent Process
In Child Process
ad@rhodes:~/ForkExecSignals-v3$
```

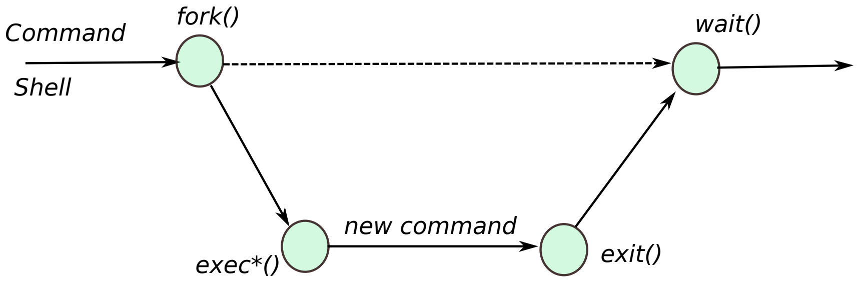
One More

- What is the output of this program and why?

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

int     main(void){

        fork();

        printf("hee\n");

        fork();

        printf("ha\n");

        fork();

        printf("ho\n");

}
```

What happens with the execution of a shell?

- A shell spawns a new process that will ultimately carry a different executable.



- The shell invokes a `fork()` creates a new process.
- The address space of the new process has to be "loaded" with a different executable with the help of `exec*()`.

The `exec*()` Family of System Calls

■ The `exec*()` family of calls provides a facility for overlaying the calling process with a new executable.

- An `exec*()`-type call NEVER RETURNS! (if all goes well)
- An `exec*()` returns -1 (ERROR), if the executable to be loaded is not found or an error has occurred.

■ There are 6 variation of `exec*()` calls that are distinguishable by the way the *command line* arguments and environment are passed and by whether a full pathname to the executable has been provided.

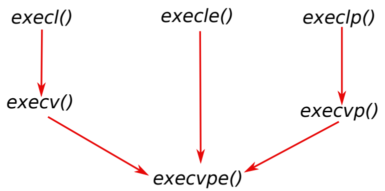
The `exec*()` Calls

- There 3 `execl*()` calls: `execl, execlp, execle`
  - they all pass the command (and its constituent elements) as a list of parameters.
  - these calls are useful if the number of command-line arguments is not known at compile time.
- There 3 `execv*()` calls: `execv, execvp, execvpe`
  - they pass their command line arguments in a *argument array* of strings.
- In above:
  - “l” stands for *list*,
  - “v” for *vector/array*,
  - “e” for *programmer constructed array*, and
  - “p” for *path* (current *PATH*).

Navigating through `exec*()` Calls

System Call	Argument Format	Environ. Passing	PATH search
`execl()`	list	auto	no
`execv()`	vector	auto	no
`execle()`	list	manual	no
`execvp()`	vector	auto	yes
`execlp()`	list	auto	yes
`execvpe()`	vector	manual	yes

- all calls do pretty much the same job with variations-all are based in `execvpe()`.



Use example for `execl()`

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

/*
 * if "cat" comes off "/bin/cat" * run the following as
 *              "5-exec-list␣5-exec-list.c"
 */

int     main(int argc, char *argv[]){

        if (argc >1){
                execlp("cat", "cat", "-T", "-n", argv[1], (char *) NULL );
                perror("exec␣failure..␣");
                exit(1);
                }
        fprintf(stderr,"Usage:␣%s␣file␣\n", *argv);
        exit(1);
}
```

Use example for `execl()`

- Execution Outcome: `./5-exec-list 5-exec-list.c`

```
ad@rhodes:~/ForkExecSignals-v3$ ./5-exec-list 5-exec-list.c
     1  #include^I<stdio.h>
     2  #include^I<sys/types.h>
     3  #include^I<stdlib.h>
     4  #include ^I<unistd.h>
     5
     6  /*
     7   * if "cat" comes off "/bin/cat" * run the following as
     8   * ^I^I"5-exec-list␣5-exec-list.c"
     9   */
    10
    11  int  ^Imain(int argc, char *argv[]){
    12  ^I
    13  ^Iif (argc >1){
    14  ^I^Iexeclp("cat", "cat", "-T", "-n", argv[1], (char *) NULL );
    15  ^I^Iperror("exec␣failure..␣");
    16  ^I^Iexit(1);
    17  ^I^I}
    18  ^Ifprintf(stderr,"Usage:␣%s␣file␣\n", *argv);
    19  ^Iexit(1);
    20  }
    21
ad@rhodes:~/ForkExecSignals-v3$
```

Use example for `execvp()`

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

/*
 * invoke this little program as: "a.out␣cat␣prog6.c"
 *
 * then try to run it as: "6-execvp-2␣cat␣-n␣6-execvp-2.c" (what happens then?)
 */

int     main(int argc, char *argv[]){

        if ( execvp(argv[1], &argv[1]) == -1 ) {
                perror("exec␣failure␣");
                exit(1);
                }
        else    {
                exit(1);
                }
}
```

Use example for `execvp()`

```
ad@rhodes:~/ForkExecSignals-v3$ ./6-execvp-2 cat -n 6-execvp-2.c
     1  #include        <stdio.h>
     2  #include        <sys/types.h>
     3  #include        <stdlib.h>
     4  #include        <unistd.h>
     5
     6
     7
     8  /*
     9   * invoke this little program as: "a.out␣cat␣prog6.c"
    10   *
    11   * then try to run it as: "6-execvp-2␣cat␣-n␣6-execvp-2.c" (what happens then?)
    12   */
    13
    14  int     main(int argc, char *argv[]){
    15
    16          if ( execvp(argv[1], &argv[1]) == -1 ) {
    17                  perror("exec␣failure␣");
    18                  exit(1);
    19                  }
    20          else    {
    21                  exit(1);
    22                  }
    23  }
    24
ad@rhodes:~/ForkExecSignals-v3$
```

Process Termination

- A process executes its last statement and then asks the operating system to delete it using the `exit()` system call.
  - If the parent used a `wait()` system call, status data about the exit of the child can return to parent process.
  - All resources of the process are *deallocated* by the kernel.
- A parent process may terminate the execution of its children.
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent exits, kernel might not allow a child to continue – *cascading termination*
  - The parent may *wait* on a child process via a `wait()` system call.

Basic Synchronization between a Parent and a Child

- With the `wait()` system call

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

int main(){
  pid_t  pid;
  int   status, exit_status;
  char *buff[2];

  if ( (pid = fork()) < 0 ){ perror("fork␣failed"); exit(1); }

  if (pid==0){
     buff[0]=(char *)malloc(12); strcpy(buff[0],"date");
     printf("%s\n",buff[0]); buff[1]=NULL;
     printf("I␣am␣the␣child␣process␣%d␣",getpid());
     printf("and␣will␣be␣replaced␣by␣'date'\n");
     execvp("date",buff);
     exit(3);
     }
  else {
     printf("Hello␣I␣am␣in␣parent␣process␣%d␣with␣child␣%d\n", getpid(), pid);
     if ( wait(&status)!= pid ) { perror("wait"); exit(1);}
     printf("Child␣terminated␣with␣exit␣code␣%d\n", status>>8);
     }
}
```

Outcome of Execution

```
ad@rhodes:~/ForkExecSignals-v3$ ./6-execvp-3
Hello I am in parent process 23208 with child 23209
date
I am the child process 23209 and will be replaced by 'date'
Tue 13 Oct 2020 11:58:33 PM EEST
Child terminated with exit code 0
ad@rhodes:~/ForkExecSignals-v3
```

■ Special status Processes: Orphans and Zombies

- If parent terminates without waiting, a child becomes an orphan.
  – Routinely adopted by "`init`" process.
- A process can find itself in a precarious position when it exits and before its return-code is ultimately read by its parent (i.e., "reaped").
  – During this *often-short time period*, an exiting process is in a limbo situation (state) and it is thus called zombie.
  – Reaping ultimately occurs when a parent *consumates* the return code of a child process.

Cooperation of Processes through *Communication*

- Cooperating processes can affect or be affected by others including their shared data space.
- Cooperation is primarily required when:
    - Information sharing among processes is a must.
    - Computations can be speed up!
    - Modularity is attained (small is beautiful etc.) and convenience
- Cooperating process need Interprocess Communication (IPC) primitives to do the work:
    - Signals
    - Pipes (regular and named)
    - Shared Memory Sergments
    - Message Passing

Signals

- Signals offer a simple method to transmit software interrupts to processes and occur asynchronously when:
  - There is an error during the execution of a job.
  - Events created with the help if input devices (*cntrl-z*, *cntrl-c*, *cntrl-*\ etc.).
  - A process notifies another one about an event.
  - Issuing of a `kill` command to a job.
- Signals are identified with integer number.
  - a unique number represent a different type of signal.

What Signals force Processes to do

- A signal forces the kernel to do one of the following things when a signal occurs:
  - Ignore the signal – two signals however can never be ignored: `SIGKILL` & `SIGSTOP`).
  - Catch the signal – we do that by informing the kernel to call a function of ours whenever a signal occurs.
  - Let the default action apply – every signal has a default action.

POSIX Signals in Linux

```
                              Action
                              -----
       SIGHUP         1       Term    Hangup detected on controlling terminal
                                      or death of controlling process
       SIGINT         2       Term    Interrupt from keyboard
       SIGQUIT        3       Core    Quit from keyboard
       SIGILL         4       Core    Illegal Instruction
       SIGABRT        6       Core    Abort signal from abort(3)
       SIGBUS         7       Core    Bus error (bad memory access)
       SIGFPE         8       Core    Floating point exception
       SIGKILL        9       Term    Kill signal
       SIGSEGV       11       Core    Invalid memory reference
       SIGPIPE       13       Term    Broken pipe: write to pipe with no
                                      readers
       SIGALRM       14       Term    Timer signal from alarm(2)
       SIGTERM       15       Term    Termination signal
       SIGUSR1       10       Term    User-defined signal 1
       SIGUSR2       12       Term    User-defined signal 2
       SIGCHLD       17       Ign     Child stopped or terminated
       SIGCONT       18       Cont    Continue if stopped
       SIGSTOP       19       Stop    Stop process
       SIGTSTP       20       Stop    Stop typed at tty
       SIGTTIN       21       Stop    tty input for background process
       SIGTTOU       22       Stop    tty output for background process
```

Action for Signals

■ The "*Action*" column above specifies the *default disposition* for each (how the process behaves when it is delivered the signal):

- `Term`: Default action is to *terminate* the process.
- `Ign`: Default action is to *ignore* the signal.
- `Core`: Default action is to terminate the process & *dump-core*.
- `Stop`: Default action is to *stop* the process.
- `Cont`: Default action is to *continue* the process if it is currently stopped.

• If any of the signals is used, the header file `<signal.h>` must be included.

Sending a `kill` System Program

- `kill [ -signal ] pid ...`   `kill [ -s signal ] pid ...`

  sends a specific signal to process(es)

```
kill -USR1 3424
kill -s USR1 3424
kill -9 3424
```

- `kill -l [signal]`: lists all available signals

```
ad@sydney:~/$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
38) SIGRTMIN+4  .........       .........
ad@sydney:~/$
```

- Sending a signal from within a process to another is done with the `kill()` sys.call:

```
int kill(pid_t pid, int sig);
```

Rewriting the handler for a signal

```
#include <stdio.h>
#include <signal.h>

void f(int);

int main(){
  int i;

  signal(SIGINT, f);
  for(i=0;i<5;i++){
        printf("hello\n");
        sleep(1);
        }
}

void f(int signum){   /* no explicit call to function f                  */
  signal(SIGINT, f);  /* re-establish disposition of the signal SIGINT   */
  printf("OUCH!\n");
}
```

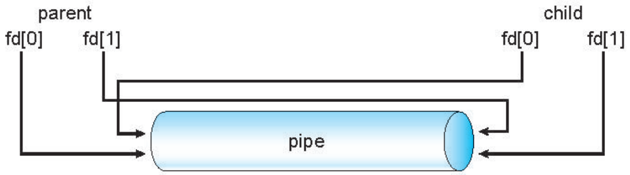
```
ad@sydney:~/$ ./a.out
hello
hello
^COUCH!
hello
hello
^COUCH!
hello
^COUCH!
ad@sydney:~/$
```

Pipes

- A pipe functions as the conduit for two processes to communicate: A process writes Bytes into the pipe and another process reads from it.
  - A pipe can be uni- or bi-directional.
  - Often pipes allows parent-child processes to communicate.
  - A regular pipe cannot be accessed from outside the process that created it.
    - Typically, a parent process creates a pipe and uses it to communicate with a child.
  - A named-pipe allow any pair of processes from the system process hierarchy to communicate through a *handle* (persistent name of a pipe).

Pipes

- Ordinary (or regular) pipes allow communication in the producer-consumer style.
  - Producer writes to one end known as the write-end of the pipe.
  - Consumer reads from the other end or the read-end of the pipe.
  - Ordinary pipes are intuitively uni-directional.
  - Ordinary pipes require a parent-child relationship between the 2 involved processes.



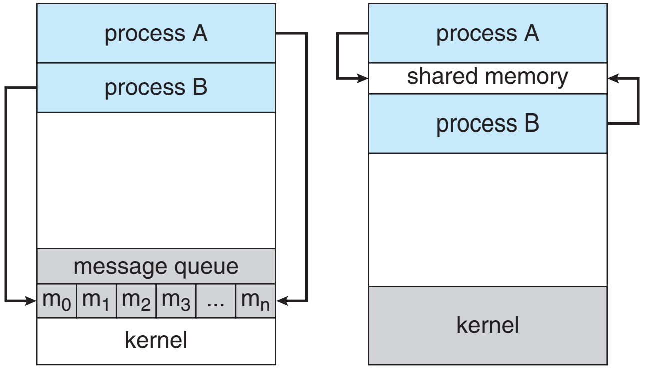
Named Pipes

- Named pipes are more versatile than regular ones.
  - Communication can be bi–directional
  - They are created, have a name (known to processes involved) and persistently “live” in the `FileSys` (they appear as files).
  - No parent-child relationship is necessary between the communicating processes
  - Several processes can use the named pipe for communication

Message Passing & Shared Memory

a) Message Passing: communication occurs using a (dual) queue of messages.

b) Shared Memory: process incorporate into their address space a *common segment of memory space*



(a) (b)

IPC - Message Passing

■ Messages can be the means for processes to communicate and synchronize their actions if needed.

- The Messaging System enable the involved processes to communicate among themselves without using *shared variable/sata-structs*.
- The IPC facility has mainly 2 operations:
  - `send(P, message)` – send a msg. to process `P`
  - `receive(Q, message)` – receive a msg. from process `Q`
- Message length can be either fixed of variable.
- Links are automatically created with the help of the kernel and each pair of processes may have uni-/bi-directional links.

IPC through Shared Memory

■ Shared Memory: an area of memory shared among the processes that wish to communicate.

- Communication is under the control of the user-processes and not the kernel (facilitator).
- Major issues is to provide mechanism that will allow user processes to synchronize their actions when they access shared memory.
- Classic synchronization example: the Producer-Consumer Paradigm.

Alex Delis, alex.delis -AT+ nyu.edu

NYU Abu Dhabi