PROCESS

AMIT KUMAR DAS

CST Department
IIEST Shibpur

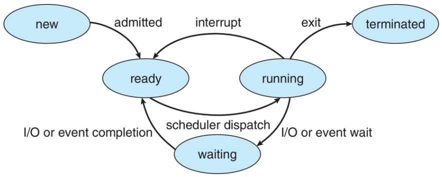
Process

- An efficient and interactive OS which should fully satisfy the user need and effectively utilize the resources of the system can only be realized through concurrency, parallelism and virtualization.
- The key to realize this goal is the *Process*; which is by definition *An instance of a program in execution*. The concept of the Process is one of the most successful ideas in computer system.

All users programs and applications need to run in the context of a process. The key abstractions that a process provides to the application are

- An *independent logical control flow* – an illusion that the program has got the exclusive use of the processor
- A *private address space* – an illusion that the program has exclusive use of the memory system.

Different states of a process in a System



- START: A new process is created and admitted to the READY state.
- READY: The process is ready to run and getting a processor shortly.
- RUNNING: The process is running in a processor.
- WAITING: wait for I/O or an event to be completed. On completion it is again ready to run
- TERMINATED: exit normally on completion or abnormally (error exit or killed)

For a better understanding of the ideas and various implementation issues involving processes requires a background on exceptional control flow (ECF).

Exceptional Control Flow: ECF

Exceptions are the basic building blocks that allow the kernel to provide the notion of a process – which by definition is *an instance of a program in execution*.

Each program in a system runs in the *context* of some process. The context consists of the state that the program needs to run correctly. The state includes

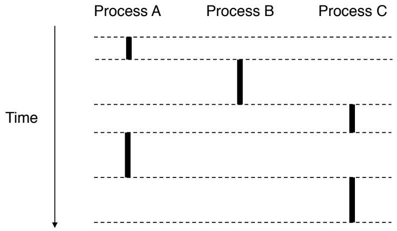
- Program's code (text) and data stored in memory
- PC and general purpose registers
- Its stack
- Environment variables and the set of open file descriptors; etc.

The key abstractions that a process provides to the application are

- An *independent logical control flow* – an illusion that the program has got the exclusive use of the processor
- A *private address space* – an illusion that the program has exclusive use of the memory system.

Logical Control Flow

Consider a system that runs three processes, A, B and C. Now, the single physical control flow of the processor can be divided into three separate interleaved logical control flow from which we may infer that



- Processes take turns using the processor
- Each process runs for while and then preempted (temorarily suspended)
- A particular process has the illusion of having the processor all the time

Concurrent Flows

Logical flow may take different forms in computer systems. Exception handlers, processes, signal handlers, threads; etc.

- A logical flow whose execution overlaps in time with another flow is known as *concurrent flow*
- And the flows are said to run *concurrently* e.g., A and B are concurrent but B and C are not.
- Multiple flows running concurrently is known as *concurrency*
- The notion of a process taking turn with other processes is also known as *multitasking*
- Each time period that a process executes a portion of its flow is called a *time slice*
- *Multitasking* is also referred to as *time slicing*

Concurrent Flows contd.

- Idea of concurrent flow is *independent* of the number of processor cores or the computers that the flows are running on
- If the two flows *overlap* in time they are concurrent even if they are running on the same processor

However, we wil sometimes find it useful to identify a proper subset of concurrent flows known as *Parallel flows*. If two flows are running concurrently on different cores or computers at the same time

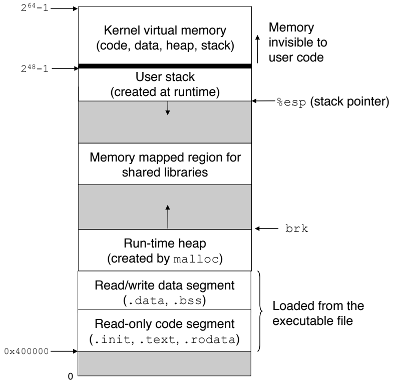
- we say that they are *parallel flows*, and
  - are running in *Parallel*, and
  - have *Parallel execution*

Private Address Space

A process provides each program with its own *private address space*; i.e.,

- It has exclusive use of the system's address space
- A byte located in an address in this space cannot be in general read or written by others

The content of the memory locations of each private space is different but their genral organisational structures are same



$2^{64}-1$
$2^{48}-1$
Kernel virtual memory (code, data, heap, stack)
Memory invisible to user code
User stack (created at runtime)
%esp (stack pointer)
Memory mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write data segment (.data, .bss)
Read-only code segment (.init, .text, .rodata)
Loaded from the executable file
0x400000
0

User and kernel mode

In order to provide the kernel a full proof process abstraction the processor must restrict the application from executing some instructions as well as the portions of the address space that it can access. This is provided by having at least two different mode of operations; namely, the *User* and the *Kernel* mode. A mode bit in a register is set/reset to move from user to kernel mode and vice-versa.

- privilege instructions like change mode bits, halt the processor or initiating I/O are not permitted in the user mode; and
- direct access to kernel to code and data in the kernel space is not allowed; any such attempt results in fatal protection fault; and
- it must access kernel code and data via the system call interface

User and kernel mode contd.

A process running an application is initially in user mode and the only way to change the mode from user to kernel is via exception; such as (i) an *interrupt*, (ii) a *fault* or (iii) a *trapping system call*
When an exception occurs, the control is passed to the exception handler and

- the mode is changed to kernel and the handler runs in kernel mode
- after the exception processing the control is passed to the application code with a switch of mode bit to user mode again.

In Linux the */proc* filesystem exports contents of many kernel data structures as a hierarchy of text files that an application can read. [Try seeing /proc/cpuinfo or /proc/process-id/maps; Now in Linux a */sys* filesystem exports additional low level information like devices and buses]

Context Switches

Kernel implements *multitasking* using a high-level form of ECF known as *context switch* – built on the lower level ECFs'. The kernel maintains a context for each process – the context is the state that the kernel needs to restart a preempted process. It contains the values of the object in the processor programming models (Registers), user and kernel stacks and various kernel data structures such as

- A *page table* – that characterises the address space; and
- A *Process table* – contains information about the current process; and
- A *File table* – contains information about the files that the process has opened

At certain points during execution of a process (running state), the kernel can decide to preempt the current process and restart a previously preempted process. This act is known as *scheduling* and handled by the kernel code, called the *scheduler*.

Context Switches contd.

When the kernel selects a new process to run we say that the process is *scheduled* – the kernel preempts the currently running process and transfers control to the new process to run by doing a *context switch* that (i) saves the context of the current process, (ii) restores the context of the previously preempted process, and (iii) passes control to this newly restored process.

A context switch can occur

- while the kernel is executing a system call on behalf of the user. If the system call blocks because it is waiting for some event to occur, then the kernel can put the current process to sleep and and switch to another process. For example for a read system call that requires a disk acess the kernel may run another process instead of the long wait for the data arriving from the disk.
- An explicit request to put the calling process to sleep using a *sleep* system call

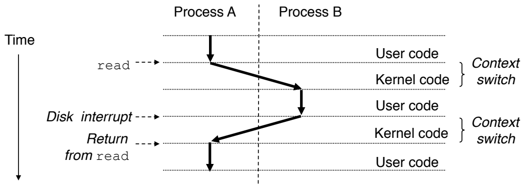
Context Switches contd.

In general, even if the system call does not block, the kernel can opt for a context switch rather than returning to the calling process.
A context switch can also occur as

- a result of an interrupt – say a timer interrupt (controlling the time slicing) deciding the current process has run for too long and chose another to run.

Next we see graphically a context switch between two concurrent processes A and B where A is running in user mode and traps to the kernel by a read system call. The trap handler in the kernel requests a DMA transfer from the disk controller and arranges an interrupt by the disk when the data transfer is done.

Context Switches Anatomy.



- the kernel was running process A in user mode before the read system call
- during the first part of the switch the kernel is executing instruction in kernel mode on behalf of process A (there is no kernel process)
- then kernel starts executing on behlf of process B (in kernel mode) and finally
- kernel is allows Process B to run in user mode
- On completion of DMA there is another context switch and finally kernel is running process A from where it left off until the next exception.

Process Control System Calls: UNIX

```
#include <sys/types.h>
#include <unistd.h>
pid_t  fork(void);  // returns 0 to Child;
// pid of the child to parent, -1 on error
```

The fork() function is used by the calling process to create a new running chid process. fork() characteristics are

- call once, return twice (always a 0 to the chid and a +ve inetger to the parent)
- concurrent execution (parent and child are concurrent processes)
- duplicate but separate address space (address space of each is separate but identical just after fork() – however with time changes are expected as they are two different processes.
- shared files (child inherits all theparent's opened files)

The charcateristics would be best understood by analysis a simple program given next [pid stands for process id and pid_t is defined as an integer]

understanding fork()

```
int main() { pid_t pid; int x = 1;
pid = Fork();  /* A wrapper for fork() explained  shorly */
     if (pid == 0){/* child */ printf("Child: x= %d\n", ++x);
             exit(0);  }
      /* Parent */
     printf(" Parent: x = %d\n", --x);
     exit(0);
}
```

Output of the program would be

```
Parent: x = 0
Child: x = 2
```

or; it could be

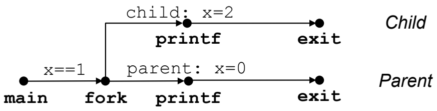
```
Child: x = 2
Parent: x = 0
```

Understanding fork()

- Call once – returns twice
  - as parent and child running concurrently – the return value of 0 to child and pid of the child to the parent helps identifying in which you are in; parent or chid
- Concurrent execution
  - Output sequence could be anything for the concurrent processes; may be different in different runs. As no ordering of execution sequence can be predicted in general.
- Duplicate but separate address space
  - After fork() – in the seprate but private address space everything is identical; so the value of x is 1 in both the cases. Finally, in parent we have decreased the value of x to 0 and in the child it has been increased from 1 to 2
- Shared files
  - Child inherits all the open files of the parent; in this case the child inherits stdout from the parent. Thus both the parent and child to produce their output to stdout; i.e., the VDU.

Understanding fork() contd.

As the fork() call is slightly tricky we may take help of a process graph as shown below graphically describing the previous program that increments a variable x in child and decrements the same in parent.

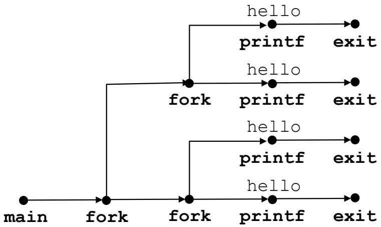


The precedence graph that captures the partial ordering of the program statement. Each vertex 'a' corresponds to the execution of a program statement. Each graph begins with a vertices representing the parent process calling main(). The arrows indicate the logical flows statement by statement up to the exit.

Program examples: Understanding fork() contd.

Take another example

```
int main() {fork(); fork(); printf("hello\n"); exit 0}
```



Here two consecutive calls to the fork() function results in four process (children) with the same code running as evidend from the graph; and we get four "hello" messages; however – the order of execution of the prcesses is random (we'll see this later through other examples)

Program examples: Understanding fork() contd.

Take another example

```
int main() { int a = 9;
  if ( fork() == 0)
printf("p1: a=%d\n", a--);
  printf("p2: a =%d\n", a++); exit (0);}
```

As fork() return 0 to the child – the first printf instruction would be executed after fork return to the child followed by 2nd printf instruction by the child again and exit. However, parent code will not reach the first printf (as pid is a positive integer) and the 2nd printf would be executed by the parent only once. So, the output would be

p1: a= 9;

p2: a= 8;

p2: a = 9; (note that the print order may change in different runs)

Using wrapper – system call error handling

Unix system calls encountering error responds by

- returning -1; and
- set the global integer variable *errno* to indicate what went wrong

PROGRAMMERS SHOULD ALWAYS CHECK FOR ERRORS.
Programmers skips error checking as it bloats the code and making it a bit uncomprehending. Here is the solution – a wrapper.

```
(A) if ((pid = fork()) < 0 ) {
fprintf(stderr, "fork error: %s\n", strerror(errno)); }
```

We can have a clean slate by using the error-reporting function

```
void unix_error(char *msg) {
 fprintf(stderr, %s: %s\n", msg, strerror(errno));
     exit(0);}  and then
 (B)    if (( pid = fork() < 0) unix_error("fork error");
```

Using wrapper – conts.

Finally, the error handling wrapper (say, Fork()) for the fork() system call

```
pid_t Fork(void)
{
pid_t pid;

if ( pid = fork()) < 0)
unix_error("Fork error");

    return pid;
}
With this wrapper, our call including the error checking is

 (C)      pid = Fork(); /* clean and unbloated */
```

For the ease of comprehension –the first letter of the function is capitalized from 'f' to 'F' – may be followed as a convention

Process Control System Calls contd.

Getting process ID – we have two call

```
#include <sys/types.h>
#include <unistd.h>
   pid_t  getpid(void);  /* returns  pid
                       of the calling process */
   pid_t      getppid(void); /* returns pid of the parent
                       of the calling process */
```

Terminating a process

```
#include <stdlib.h>
   void  exit(int status); /* you may return an exit
                                 status to the caller */
```

```
 The other ways to terminate a process is through a
    signal or when you return from the main
```

Reaping a child process

When a process terminates the

- kernel does not remove it immediately from the system.
- The kernel waits until it is reaped by its parent
- When it is reaped by its parent the kernel returns the exit status to its parent
- And discards the terminated process – and then it does not exist

A terminated process yet to be reaped by its parent is known as a *Zombie*; i.e., a living corpse as depicted in folklores. When a parent process terminates, the kernel arranges for the init process (the ancestor of every process created by the kernel with pid 1 and that never terminate) to adopt the orphaned children. And finally the kernel arranges for the init process to reap them.

Reaping a child process contd.

waitpid() function

```
#includes  <sys/types.h>
#include <sys/wait.h>
pid_t  waitpid(pid_t pid, int *statusp, int options);
/* returns PID of the child; 0 (if WNOHANG) & -1 on error  */
```

By default (options = 0) *waitpid* suspends execution of the calling process until a child process in its wait set terminates. If a process in the wait set has already terminated – waitpid returns immediately. The parent gets the pid of the terminated child that causes the waitpid to return.

Reaping a child process contd.

Members of the wait set can be determined by the pid argument.

- if pid $>$ 0; wait set is the singleton child process whose pid is passed.
- if pid $=$ -1; wait set consists of all of the child processes of the parent

On error (say, the caller has no child) it returns -1 and sets *errno* to ECHILD. If waitpid is interrupted by a signal (more later) it sets the *errno* to EINTR.

wait() function

It is a simpler version of the complicated *waitpid()* function

```
#includes  <sys/types.h>
#include <sys/wait.h>
pid_t  wait(int *statusp);
/* returns PID of the child if OK or  -1 on error */
```

Calling wait(&status) is equivalent to calling waitpid(-1, &status, 0).

Program examples: Understanding fork() and waitpid()

```
int main(){
if (fork() == 0) {printf("9"); fflush(stdout);}
else {printf("0"); fflush(stdout);
    waitpid(-1, NULL,0);
    }
printf("3"); fflush(stdout);
printf("6"); exit(0);
}
```

What is the output of this program? Try predicting the output using the precendence graph.

Putting Processes to Sleep and Pause

The **Sleep** and **Pause** function suspends a process for a specified period of time.

```
#include <unistd.h>
unsigned in sleep(unsigned int sec);
/* returns: 0 if time is already elapsed else  seconds left to
to make a prematured exit */
```

The **pause()** function is useful which puts a proces to sleep until it receives a signal.

```
int pause(void);  /* always returns -1 */
```

A wakeup wrapper for sleep function that prints a mesage on wake up.

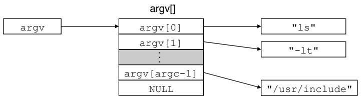
```
unsigned int wakeup(unsigned int secs)
{unsigned int rc = sleep(secs);
  printf("Woke up at %d secs.\n", secs - rc +1); return rc;}
```

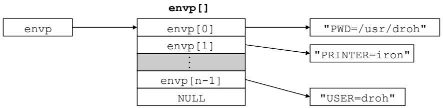
Loading and running a program in the context of a process

Execve() function is used to load and run users program.

```
#include <unistd.h>
int execve(const char *filename, const char *argv[],
     const char *envp[]);  /* Returns -1 on error
                              otherwise does not  return */
```

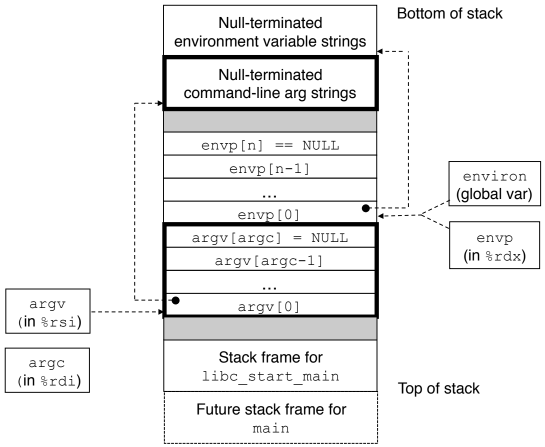
Organisations of the argument and environment variable lists





After loading the program execve calls the start up code which sets up the stack and passes control to the main routine having the form shown below
int main(int argc, char *argv[], char *envp[]);

titleUser's stack at start



libc_start_main is the startup code that calls main(). Note that register rsi and rdx (x86-64 registers) point to the argv and envp strings while argc is available in rdi.

Running a Program: fork() and execve()

- Unix shell and Web servers make frequent use of the fork() and execve() functions to run programs
- The shell is an application level program that runs another program on behalf of the user using the following steps
    - Reads a Command Line from the user
    - Parse the command line; and
    - Runs the program on behalf of the user

Next is an example of a main() routine of a simple shell which prints a prompt and waits for user input through *stdin* and then evaluates the command line to take action

Running a Program contd.

```
#define MAXARGS 128   // appropriate .h files to
// be included before this
void eva(char *cmdline); // function prototypes
int  parseline(char *buf, char *argv[]);
int builtin_command(char *argv[]);


int main() { char cmdline[MAXLINE];
while(1){ printf("Ready>");
       fgets(cmdline, MAXLINE, stdin); // Read commandline
     if ( feof(stdin))
exit(o);
eval(cmdline); /* evaluate the commandline
```

The next is the Evaluate routine

Running a Program contd.

```
void eva(char *cmdline){
char *argv[MAXARGS]; // Argument list execve()
char buf[MAXLINE]; // holds modified command line
int bg; /* background job? */ pid_t pid // process id

strcpy (buf, cmdline); bg = parseline(buf, argv);
   if (argv[0] == NULL) return;
    if (!builtin_command(argv)){// expecting users executable
if (( pid = Fork()) == 0) { // let the child runs user program
if (execve(argv[0], argv, environ) < 0) {
printf("Command not found\n"); exit(0); } }
if(!bg) {
int status;
if (waitpid(pid, &status, 0) < 0)
unix_error("waitpid error"):
     } else printf("%d %s ", pid, cmdline); } return; }
```

Running a Program contd.

If the first argument is a bult-in comman – run it

```
int builtin_command(char *argv[])
{
if (!strcmp(argv[0], "quit")); exit (0);
   if (!strcmp(argv[0], "&")); return 1;

   return 0; // not a built-in command
}
```

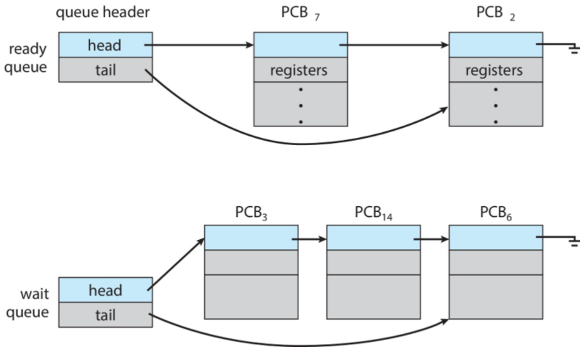
Running a Program contd.

```
int parseline(char *buf, char *argv[]){ // parse command line
 and build argv array
char *delim; /* points to first space delim.*/ int argc; int bg;
buf[strlen(buf) - 1] = ' '; // replace trailing '\n' with space
while(*buf && (*buf == ' ')) buf++ // skip past leading spaces
argc = 0; // build argvs'
 while ((delim = strchr(buf,' '))){ argv[argc++]=buf;*delim = '\0';
buf = delim + 1;
 while(*buf && (*buf == ' ')) buf++;} // skip past trailing
spacesargv[argc] = NULL;
    if (argc == 0) return 1; // ignore blank line
   if (( bg = (*argv[argc-1] == '&')) != 0) // should job in
 the bg mode
argv[--argc] = NULL;
    return bg;
}
```

Program Vs. Process

- A program
  - is a collection of code and data and
  - may be stored as an object file on the disk or as a segment in an address space
- A process
  - is an instance of a program in execution and
  - a program always runs in the context of a process
- The fork() function creates a new process (the child) and
  - runs the same program in a new child process; and
  - this child process is a duplicated of the parent
- The execve() function does not create a new process rather
  - it loads and runs a new program in the context of a current process by overwriting the address space of the current process; and
  - the new program still has the same pid and enjoys the inherietence of all the file descriptors that were open at the time of calling execve

Ready and Wait Queues


queue header
PCB 7
PCB 2
ready queue
head
tail
registers
registers
PCB3
PCB14
PCB6
wait queue
head
tail