# Operating Systems & Concurrency: Process Concepts

Michael Brockway

October 6, 2011

# Outline

- Processes - context, data area, states
- Process creation, termination – unix examples
- Processes and threads

# Processes

Definition: A *process* is an *instance* of a program (an application) that is *running* under the managment of the operating system.

A process runs *sequentially*

Modern operating systems *multitask* – they manage the simultaneous running of several (lots!) of application processes. The management is performed by a *scheduler*.

# Processes

Each process has

- a process *ID*
- a *priority*
- its own *context* and *state* (see below)
- CPU *scheduling* information
- file handles, network ports; I/O status information
- a data area; memory management information
- etc

These data form a the fields of the *process control block* (PCB), aka *task control block* (TCB) for the process/task. The operating system keeps a table of PCBs for all of the currently executing processes.

# Process context

This is the set of values in all the *CPU registers* including

- the *program counter*: the address in memory from where the next instruction will be fetched
- the *program status register*: bits are set according to the result of the last operation; the current operation may test these bits
- the *stack pointer*: the address of the current top of the stack
- one or more data registers (*accumulators*) for holding data temporarily during a computation

Subroutine calls are normally managed by storing return address, parameters, return value(s) on the *stack*

# Process data area

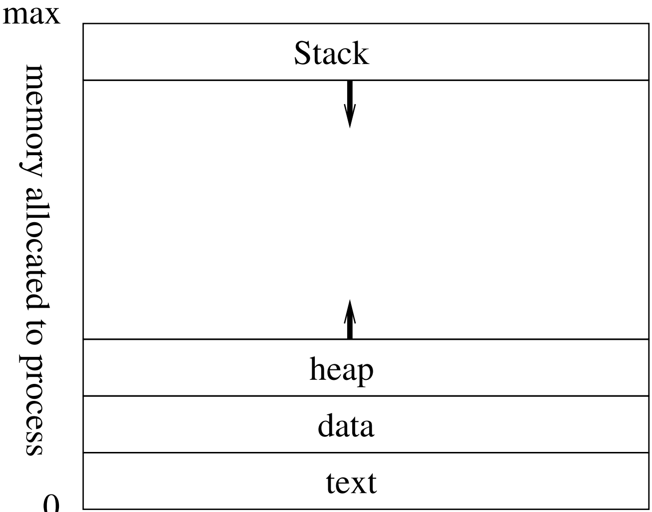


Figure: Data area for a process

# Process states

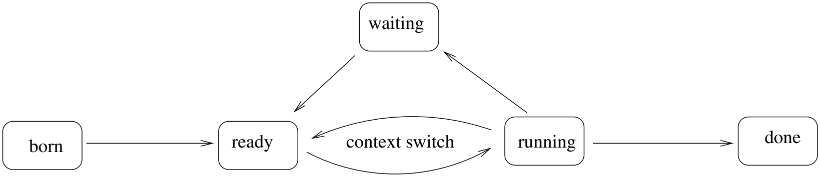


Figure: Life cycle of states for a process

In a *multitasking* system, there will be several (lots) of tasks (processes) "running". One will be actually *running* on the CPU; the others will be *ready*. The *scheduler* periodically performs a *context switch* to give all the tasks a turn. Ususally this is rapid, giving the appearance of simultaneously, *concurrently* running processes.

The processes/tasks are *logically* concurrent.

If there are multple CPUs, there will be mutlple *running* tasks – 1 per CPU. In this case we have some actual *physical* concurrency.

# Process creation

A “parent” process create “children” processes, which, in turn create other processes, forming a tree of processes.

Generally, a process is identified and managed via a *process identifier* (pid)

Resource sharing

- Parent and children share all resources; or
- Children share a subset of the parents resources; or
- Parent and child share no resources

Execution

- Parent and children execute concurrently; or
- Parent waits until children terminate

# Process creation - UNIX example

The *fork* system call creates a new process

The *exec* system call used after a fork to replace the process memory space with a new program

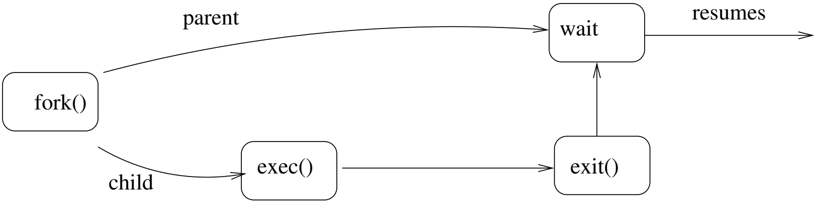


Figure: Fork exmaple

# Process creation - UNIX example

```
int main() {
pid_t  pid;
  pid = fork();              /* fork another process */
  if (pid < 0) {             /* error occurred */
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }
  else if (pid == 0) { /* child process */
    execlp("/bin/ls", "ls", NULL);
  }
  else { /* parent process */
    /* parent will wait for the child to complete */
    wait (NULL);
    printf ("Child Complete");
    exit(0);
  }
}
```

# Processes termination (UNIX)

Process executes last statement and asks the operating system to delete it (exit)

- Output data from child to parent (via wait)
- Process resources are deallocated by operating system

Parent may terminate execution of child processes (abort)

- if child has exceeded allocated resources, or
- task assigned to child is no longer required, or
- if parent is exiting

Some operating systems do not allow child to continue if its parent terminates: all children are terminated - *cascading termination*.

# More on `fork`

Notice from the code that `fork` returns an integer value -

- 0 if it is in the parent process
- a nonzero value in the child (clone) – in fact, the process ID of the parent.

If exec() is not used, `fork` simply creates a "clone" of the parent process: global variables, file handles etc are duplicated. Can you explain the behaviour of the following program (`processdemo.c` in Source files set 1, downloadable from here)?

```
#include <stdio.h>
int x = 50;   /* a global variable */
```

# More on fork

```
void adjustX(char * legend, int i)
{  long p;
   while (1)  /* loop forever */
   {   printf("%s: %i\n", legend, x);
       x += i;
       p=0;
       while (p<100000000) p++;  /* a "busy" delay */
   }
}

main()
{  int c;
   printf("creating  new process:\n");
   c = fork();
   printf("process %i created\n", c);
   if (c==0)
      adjustX("child", 1);    /* child process */
   else
      adjustX("parent", -1);  /* parent process */
}
```

# Processes and Threads

A “simple” process executes *sequentially* – one operation at a time: a *single-threaded process*

A process *may* be *multithreaded*

- Several threads, *each one* executes sequentially
- Each thread is scheduled as it if it were a separate process
  - Each thread has its own subroutine stack
  - Each thread has a distinct state, its own scheduling information
- But the threads belonging to a particular process share all other data, memory management information, file handles, network ports, I/O status information

A thread is a “light-weight” process-like entity; *part of* a process.

Normally a process is ended when all its threads have ended.

- A *daemon* is a thread which carries on executing after its parent process has finished.

# UNIX (POSIX) Threads example: threaddemo.c

The source file is in "Source files set 1", downloadable from here.

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

int x = 50;   /* a global (shared) variable */

void * adjustX(void *n)
{  int i = (int)n;
   long p;
   while (1)   /* loop forever */
    {   printf("adjustment = %i; x = %i\n", i, x);
        x += i;
        p=0;
        while (p<100000000) p++;    /* a "busy" delay */
   }
   return(n);
}
```

# UNIX (POSIX) Threads example

```
main()
{  int a;
   pthread_t  up_thread, dn_thread;

   pthread_attr_t *attr;  /* thread attribute variable */
   attr=0;

   printf("creating threads:\n");
   pthread_create(&up_thread,attr, adjustX, (void *)1);
   pthread_create(&dn_thread,attr, adjustX, (void *)-1);

   while (1) /* loop forever */
   { ;}
}
```