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

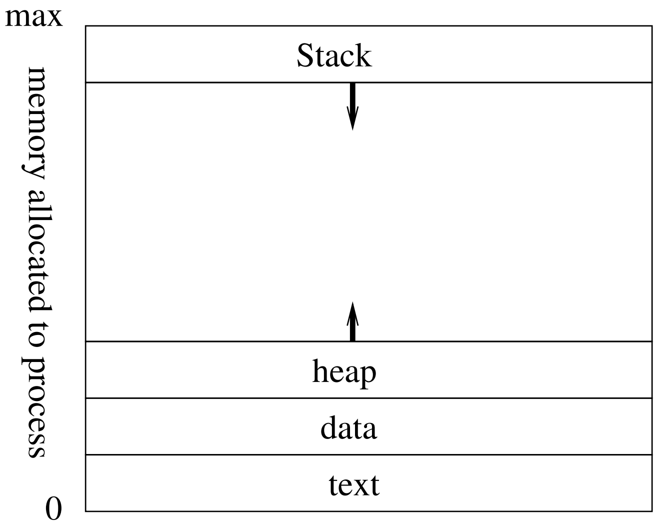


Figure: Data area for a process

Process states

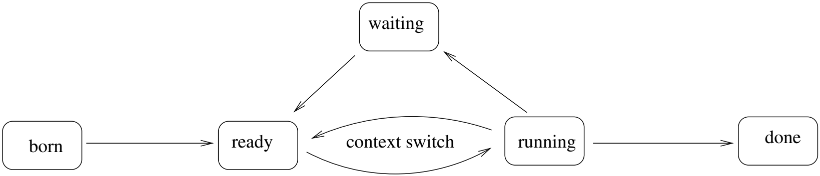


Figure: Life cycle of states for a process

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

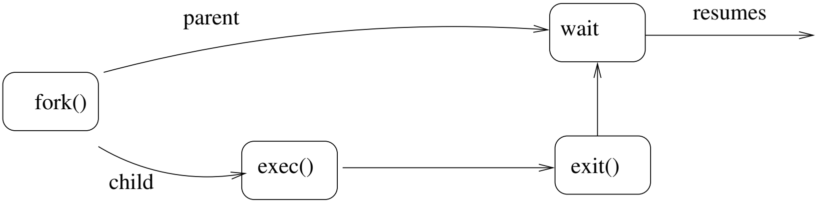


Figure: Fork exmaple

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 */
   { ;}
}
```

Operating Systems & Concurrency: Process Scheduling and Communication

Michael Brockway

October 6, 2011

Scheduling

The mix of running processes is managed by a *scheduler* process which gives the "running" processes turn-about on the CPU.

- This may be a straight "round-robin"; or
- it may be that processes are assigned *priorities*:
  - A higher priority process is given the CPU ahead of a lower-priority process.

The scheduler manages a set of processes/tasks that are *ready to run*. One of them is currently *running*. In a *context-switch* performed by the scheduler the running task changes places with one of the ready tasks.

This happens frequently (commonly 20-50 times per second) so that the processes/tasks appear to be running simultaneously.

Scheduling

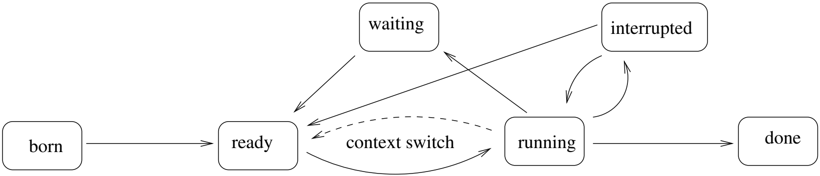


Figure: States of a process

In reality a process passes between a number of *states*, doing many *context switches* before finishing.

When the process requires an operating system *resource* (graphics, disk or network I/O, or a communication from another process) in order to complete an operation it may have to *wait*.

The scheduler moves it from the *running* state to the *waiting* state. Eventually a waiting task "wakes up"– when it has the resource, commmunication or whatever it was waiting for – and return to the *ready* state: it is elegible again for context switch into *running* state.

Scheduling - process context

Each process has its own *context*:

- program counter;
- program status register;
- other hardware *registers*

Recall the *fetch-execute cycle* -

1. using address in program counter, fetch next instruction from memory;
2. increment program counter by size of instruction;
3. decode the instruction
4. execute the instruction
5. check program status register and possibly reload program counter
6. check for an *interrupt*

Scheduling – interrupts

The CPU can be *interrupted* by an event in its environment – a signal from a peripheral. It handles the interrupt by

1. saving the process context;
2. looking up the address of the interrupt's *handler* routine in the *interrupt vector table*;
3. running the handler
4. restoring the process context - this resumes the process

In a multitasking environment there is a *timer* which periodically fires an interrupt. The handler determines using a *scheduling algorithm* whether the currently running task has had a long enough turn on the CPU and if so, effect a context-switch: restore the context of *another* ready task. The former running task goes to the back of the ready queue: the dotted arrow in figure 1) is thus effected two transitions via the *interrupted* state.

In general the processes/tasks in the ready and waiting states are in *prioritized queues*.

Scheduling - flow of states

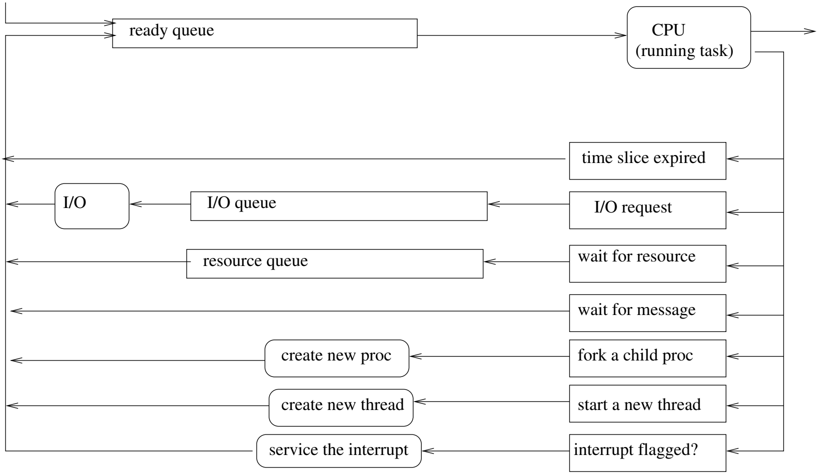


Figure: “Life cycle” flow

Scheduling activities

Long-term scheduling

- repeat period = seconds or minutes
- admits new processes to ready queue

Short-term scheduling

- repeat period = milliseconds
- triggers a context switch

I/O can take a long time – longer than the short-term scheduling interval. In this case the process is said to be *I/O-bound*.

At the opposite extreme is a process which spends most of its time performing computations - a *CPU-bound* process.

Scheduling Algorithms

How does the scheduler decide which process to run? Aim to -

- be fair – give all processes a fair amount of CPU time
- minimize response time
- minimize turn around time
- maximize CPU utilization
- meet user deadlines
- maximize system utilization

These can't ALL be satisfied!

Types of Scheduling Algorithms

A *non-preemptive* scheduler allows the *running* task to continue running until it gives up the CPU: changes state because it is *waiting* for a resource or message. Examples -

- first-come-first-served
- shortest-job-first

With a *preemptive* scheduler, the tasks/processes all have priorities assigned and the *running* task is switched out to *ready* as soon as a higher-priority task becomes *ready*. If its time-slice is exhausted before this happens another equal-priority task may be switch-in.

- Priorities are be assigned statically or dynamically
- "Round-robin" scheduling fits within this approach

Scheduling Examples

Scenario: 3 jobs:

1. `loop (3ms CPU, 3 ms I/O time) 6 times`
2. `loop (1ms CPU, 5 ms I/O time) 6 times`
3. `loop (8ms CPU, 1 ms I/O time) 4 times`

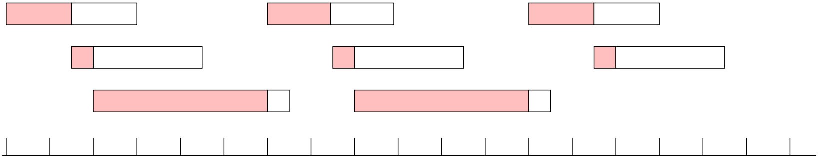


Figure: First-come first-served

(shaded = CPU time, unshaded = I/O pending)

Scheduling Examples - ctd

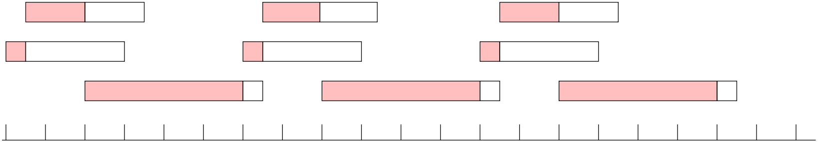


Figure: Shortest job first

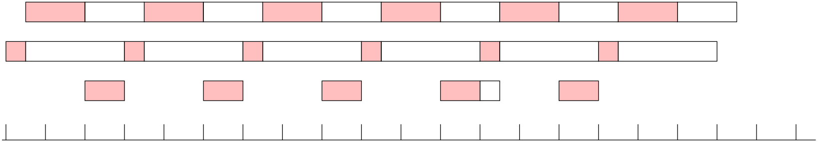


Figure: Pre-emption, I/O-bound jobs have priority

Scheduling Examples - ctd

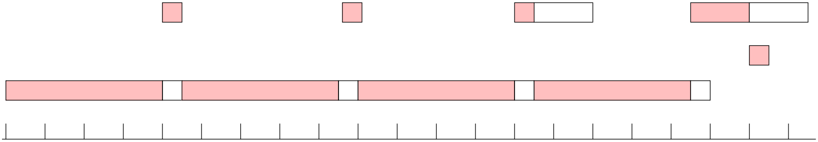


Figure: Pre-emption, CPU-bound jobs have priority

Scheduling - priorities

Priorities can be assigned to jobs *statically*

- In MicroC, a *task* is assigned a priority when it is created.

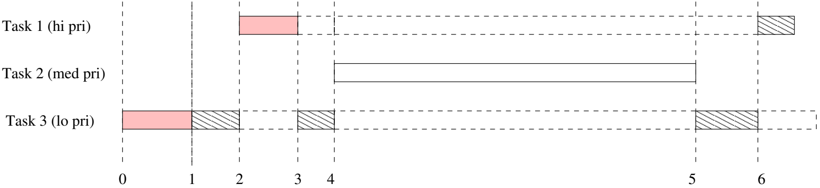
A option is to be able to change a task priority even when it is running.

Algorithms exist for assigning priorities for example -

- A *periodic* task has a "duty cycle", a loop which repeats at a fixed interval. *Rate-monotonic scheduling* assigns priorities in *increasing* order of frequency – more frequently recurring tasks are given higher priority.
- A set of tasks have have *deadlines* defined. *Deadline-monotonic scheduling* assigns priorities in order of deadlines, the earlier the deadline, the higher the priority.
- A priority may be adjusted according to the "age" the

Priority Inversion

An example of what can go wrong! Consider three tasks (low, medium, high priority) and a shared resource, a data file being written to.



0. Initially the low priority task, 3, is running by itself ...
1. After a while it obtains an exclusive lock on a data file ...
2. Then high-priority task 1 becomes ready and pre-empts task 3 ...
3. But then it wan't the file locked by 3, so must wait. 3 resumes ...
4. Task 2 (medium pri) pre-empts task 3 which still has file locked ...
5. Task 2 finishes, task 3 resumes. Task one still waiting for file.
6. Task 3 releases lock on file, so task 1 can obtain it and resume. Task 1, *high priority* has been kept waiting a long time by the lower prority tasks!

Communication betweeen Tasks

Processes within a system may cooperate: communicate or synchronise, affect one aanother, share data. Reasons include

- Information sharing
- Computation speedup
- Modularity
- Convenience

Interprocess communication (IPC) may be by *shared memory* or by *message passing*

Communication by Shared Memory

Shared memory may use a *bounded buffer*

- a shared variable, or
- a “circular” array
  - Indexes are incremented and wrap around when they reach the end

In case such as these *synchronisation* is required to prevent a writer process overwriting the buffer before a reader process has read data there, and to prevent a reader reading data already read, before it is refreshed.

- The producer-consumer problem – see `prodconsUnsync.c` in Source files set 1, downloadable from here

Communication by Message Passing

May be

- Synchronous (*blocking*) – sending (receiving) process *waits* until receiving (sending) process has "synchronised" or "rendezvoused";
- Asynchronous (*non-blocking*) – eg
  - sending process puts message in a *message queue* or *mail box*
  - receiving process takes message from queue or mail box
  - sender may *wait* if queue/mail box full;
  - receiver may *wait* if queue/mail box empty.
- Unicast, multicast, broadcast
- unidirectional, bidirectional

You will apply a number of these techniques for synchronisation (including Dijkstraś *semaphores* and *mutexes*) and message passing in real-time embedded systems.