Lecture 2

Processes, threads, scheduling

Process structure

- There are user processes and OS processes.
- The process consists of the executable (instructions), its data, stack, other buffer memory and administrative (management, part of the context) information.
- The administrative information is mostly stored in the kernel space.

Main component of the process context - process descriptor

- Process ID: unique integer value
- Parent process ID
- Real user ID: the id of the user who started this process
- Effective user ID: user whose rights are carried (normally the same as above)
- Current directory: the start directory for looking up relative pathnames
- File descriptor table: table with data about all input/output streams opened by the process. It is indexed by an integer value called file descriptor.
- The environment: list of strings VARIABLE = VALUE used to customize the behaviour of certain programs.

- Code area
- Data area
- Stack
- Heap
- Priority
- Signal disposition: masks indicating which signals are awaiting delivery, which are blocked.
- Umask: mask value used to ensure that specified access permissions are not granted when this process creates a file.

Process management

- Create: the internal representation of the process is created; initial resources are allocated; initialize the program that is to run the process.
- Terminate: release all resources; possibly inform other processes of the end of this one.
- Change program: the process replaces the process it is executing (by calling exec).
- Block: wait an event, e.g., the completion of an I/O operation.
- Awaken process: after sleeping, the process can run again.
- Switch process: process context switching.
- Schedule process: takes control of the CPU.
- Set process parameters: e.g., priority.
- Get process parameters: e.g., CPU time so far

Create a child process

- A process can create a child process, identical to it, by calling fork() – Unix function. As the kernel creates a copy of the caller, two processes will return from this call.
- The parent and the child will continue to execute asynchronously, competing for CPU time shares.
- Generally, users want the child to compute something different from the parent. The fork() returns the child ID to the parent, while it returns 0 to the child itself. For this reason, fork() is placed inside an *if test.*
- Example:

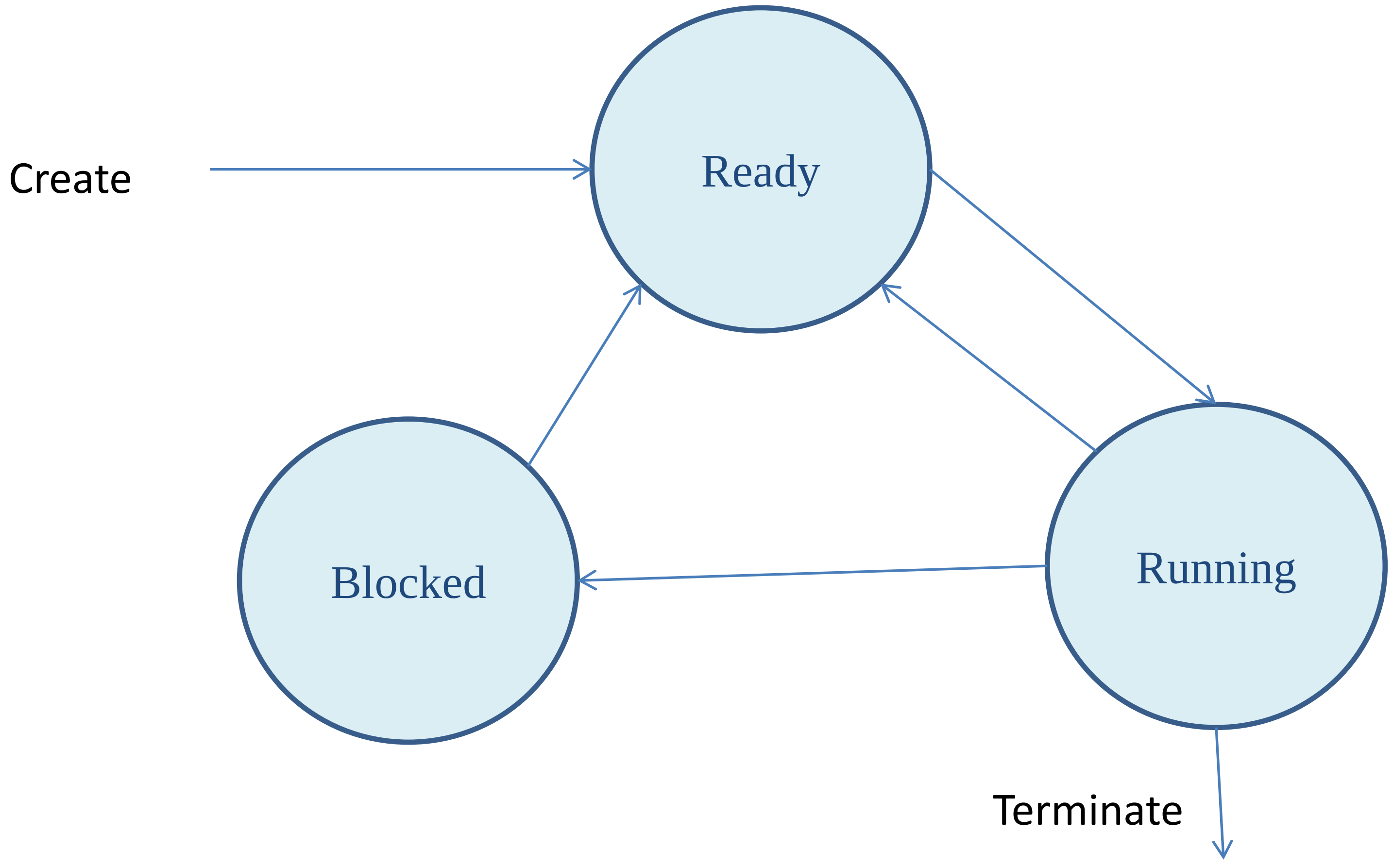
```
int i;
if (fork()) { /* must be the parent */
          for (i=0; i<1000; i++)
                    printf("\t\t\tParent %d\n", i);
          }
          else { /* must be the child */
                    for (i=0; i<1000; i++)
                              printf("Child %d\n", i);
          }
```

- Question: in what order will the two strings be printed ?

Process genealogy

- All processes are descendents of the init process, whose PID is 1.
- The kernel starts init in the last step of the boot process.
- The init process, in turn, reads the system init-scripts and executes more programs, eventually completing the boot process.
- Every process on the system has exactly one parent.
- Likewise, every process has zero or more children.
- Processes that are all direct children of the same parent are called siblings.

Basic process states



Thread

- A thread is known as a lightweight process; within a process we can have one (process ≡ thread) or more threads.
- All threads share the process context, including code.
- The context private to each thread is represented by the registers file and stack, the priority and own id.
- Generally the thread switch within the process is handled by the thread library, without calling the kernel. It is very fast as the thread context is minimum.
- When a process starts execution, a single thread is executed, which begins executing the main() function of the program. It will continue so until new threads are created:

```
thread_create(char *stack, int stack_size, void (*func)(), void *arg);
```

Advantages/disadvantages

- Threads provide concurrency in a program. This can be exploited by many-core computers.
- Concurrency corresponds to many programs internal structure.

- If a thread changes directory, all threads in the process see the new current directory.
- If a thread closes a file descriptor, it will be closed in all threads.
- If a thread calls exit(), the whole process, including all its threads, will terminate.
- If a thread is more time consuming than others, all other threads will starve of CPU time.

Purpose of scheduling

- Historically, the CPU was allocated to one process until its completion – known as batch processing. Then, the CPU was time-shared by multiple processes ready to execute.
- As CPU is time-shared, processes compete for the next available time slot.

- The scheduler is the kernel process that implements an algorithm that decides which process gets the CPU next.

- The scheduling process needs to be fair to all processes.
- Processes ready to execute are organized in a queue from where the scheduler selects the next one.
- A process takes control of the CPU by having its state restored, after the state of the previous process is saved.

Scheduling strategies: first-come first-served / round-robin

- FCFS is the simplest algorithm: processes are getting CPU control in their order in the ready-to-execute queue.
- One possibility is to have the control of the CPU until the process finishes - non-preemptive execution. This may lead to starvation of other processes.
- Therefore, the best solution is to time-share the CPU.
- If a process is not finished during its time slice, it will be returned at the end of the queue.

- Other possibilities to be switched from the running state are:
  – an I/O operation that will put the process in the blocked queue;
  – it suspends itself until a certain event occurs;
  – a higher priority process requires control.

Example: shortest process first

- If the CPU is not time-shared, the order in which processes are scheduled is important.
- Processes can be ordered according to their execution time.
- If processes get control in the increasing value of their execution time, the average turnaround time is better than in the random order.
- The *turnaround time* is the time consumed from the moment the process is ready for execution until its completion.

- Example: 3 processes, a(40), b(60), c(20); Tat: average turnaround time

Ta = 40, Tb = 100, Tc = 120 Tat = (Ta + Tb + Tc)/3 = 260/3

In increasing value of execution time:

Tc = 20, Ta = 60, Tb = 120, Tat =200/3

Conclusions

- Processes and threads are the main execution abstractions managed by the kernel.
- Thread switch is much faster than the process switch.
- The scheduler is the kernel process that decides which process will take control of the CPU in the next time slice.